Titre: Title:	Une nouvelle perspective pour gérer le soutien de l'utilisabilité dans l'architecture logicielle
Auteur: Author:	Tamer Rafla
Date:	2005
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Rafla, T. (2005). Une nouvelle perspective pour gérer le soutien de l'utilisabilité dans l'architecture logicielle [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/7674/

UNIVERSITÉ DE MONTRÉAL

UNE NOUVELLE PERSPECTIVE POUR GÉRER LE SOUTIEN DE L'UTILISABILITÉ DANS L'ARCHITECTURE LOGICIELLE

TAMER RAFLA

DÉPARTEMENT DE GÉNIE INFORMATIQUE

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION

DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES

(GÉNIE INFORMATIQUE)

NOVEMBRE 2005

Your file Votre référence
ISBN: 978-0-494-16843-1
Our file Notre référence
ISBN: 978-0-494-16843-1

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

UNE NOUVELLE PERSPECTIVE POUR GÉRER LE SOUTIEN DE L'UTILISABILITÉ DANS L'ARCHITECTURE LOGICIELLE

présenté par: RAFLA Tamer

en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées

a été dûment accepté par le jury d'examen constitué de:

M. GRANGER Louis, M.Sc., président

M. ROBILLARD Pierre-N, Ph.D., ing, membre et directeur de recherche

M. DESMARAIS Michel, Ph.D., membre et codirecteur de recherche

M. SEFFAH Ahmed, Ph.D., membre

A ma famille pour leurs encouragements continus

REMERCIEMENTS

Je tiens d'abord à exprimer mes sincères remerciements à mes codirecteurs de recherche MM. Pierre N. Robillard et Michel Desmarais, pour avoir patienté avec moi au cours de ce travail de recherche et pour tout le temps qu'ils ont mis à ma disposition. Ils m'ont guidé avec attention, efficacité et ouverture, tout en m'accordant initiative et autonomie.

Je tiens aussi à remercier le personnel technique du département : Jean-Marc Chevalier, Louis Malo et Francis Gagnon pour leur patience, leur gentillesse et leur disponibilité. Merci également à mes collègues du Laboratoire de Recherche en Génie Logiciel (RGL) avec qui j'ai eu des échanges aussi nombreux que fructueux. Leurs amitiés ne seront jamais oubliées.

Un merci va à tous mes proches, amis (es) et parents, qui depuis le premier jour ont cru en moi et m'ont encouragé à toujours donner le meilleur de moi-même. Je suis reconnaissant à ma famille pour tous les sacrifices qu'ils ont faits pour moi. Je ne peux terminer ces lignes sans remercier profondément mon frère, Ramez, pour son soutien permanent et son encouragement.

Je ne peux compléter mes remerciements sans mentionner Shérif Ebeid, Mina Mikail et Martin Isaac qui m'ont beaucoup aidé avec le formatage et la révision du mémoire. Un grand merci!

Je remercie enfin toutes les personnes intéressées par mon travail, en espérant qu'elles puissent trouver dans mon mémoire des explications utiles pour leurs propres travaux.

RÉSUMÉ

Les ingénieurs logiciels font aujourd'hui face à d'énormes pressions pour développer et déployer des systèmes informatiques complexes. Ces pressions ont eu pour effet la concentration de leurs efforts dans le développement des besoins fonctionnels du client. On se rend compte aujourd'hui que ceci a eu pour conséquence l'oubli de certains attributs importants de qualité, surtout l'utilisabilité, entraînant alors la livraison de systèmes informatiques de mauvaise qualité. Jakob Nielsen, un gourou de l'utilisabilité, affirme qu'un logiciel qui fournit à l'utilisateur une panoplie de fonctionnalité mais qui est difficile à utiliser ne pourrait se vendre. Ceci va à l'encontre d'un des objectifs majeurs du génie logiciel : la conception de systèmes que les utilisateurs trouveront utilisables et qui leur permettront d'accomplir d'une manière efficace leurs travaux. Il est donc indispensable de prendre en compte l'utilisateur et son contexte de travail dans la conception de systèmes informatiques. Le succès d'un logiciel ne dépend donc plus uniquement de sa fonctionnalité ou de son prix, mais surtout de sa capacité à être utilisé facilement par une large population de connaissances variables.

L'utilisabilité n'est généralement pas considérée pendant la conception de l'architecture, principalement à cause de la croyance qu'elle fasse seulement partie de l'Interface Utilisateur (IU). Mais la modification de certains aspects esthétiques de l'IU peut avoir un impact sur l'architecture du système. Plus précisément, ils ne peuvent être rectifiés sans encourir des changements majeurs à l'architecture de l'application.

Seulement deux groupes de recherche ont proposé des méthodologies pour élucider le lien qui existe entre l'utilisabilité et l'architecture. Ils ont identifié des exigences d'utilisabilité qui nécessitent une considération lors de l'élaboration de l'architecture. Il s'avère très difficile d'incorporer ces exigences dans un système existant si leur absence n'a été découverte qu'à la livraison du système au client. Ceci nous mène à conclure que plus d'attention devrait être consacrée aux méthodes

d'analyse des architectures qui seront orientées utilisabilité. De ce fait, nous présentons une étude de cas qui consiste à adapter une méthode existante d'analyse de l'architecture (SAAM) afin d'identifier l'impact de mettre en application ces changements d'utilisabilité sur l'architecture. Nous analysons une application Web, développée par des étudiants de 4ème année en Génie Logiciel à l'École Polytechnique de Montréal, afin de déterminer comment l'utilisabilité peut être incorporée. Une analyse détaillée nous permet d'affirmer que l'inclusion de certaines des exigences d'utilisabilité n'a été possible sans encourir des changements architecturaux majeurs. Corriger les problèmes d'utilisabilité sur un système existant peut s'avérer coûteux si les changements exigent les modifications qui atteignent au-delà de la couche de présentation, plus précisément ceux qui ne peuvent pas être facilement accommodés par l'architecture de logiciel.

En tenant compte des exigences d'utilisabilité plus tôt dans le cycle de développement de logiciel, plus spécifiquement avant la phase de conception, le coût de ces modifications peut être considérablement réduit. Cependant, il y a une pénurie de méthodes et de directives qui ont pour but de guider les intervenants dans l'obtention des requis d'utilisabilité qui peuvent avoir un sérieux impact sur l'architecture de logiciel. Nous proposons donc une méthode qui va aider les organisations qui développent des logiciels, à découvrir et documenter les requis d'utilisabilité. Une expérience empirique, entreprise pour évaluer l'utilité de cette méthode, indique que la méthode permet de discerner les aspects de rentabilité qui n'auraient pas été nécessairement définis si cette technique n'avait pas été utilisée. Ensuite, nous modéliserons cette méthode comme activité au sein d'un processus centré architecture et nous définirons les rôles ainsi que les artefacts qui seront associés à cette activité.

ABSTRACT

In the last decades, it has become clear that the most challenging task for a software engineer is not just to design for the required functionality, but also focus on architecting for specific quality attributes such as performance and security, which contribute to the quality of the software. Software that provides much functionality but is awkward to use will not sell. Over the years it is noticed that besides an increasing focus on quality attributes, increasing attention is being paid to the architecture of a software system. And within the software engineering community, it is commonly accepted that the quality attributes of a system such as performance or security are to a large extent, constrained by its architecture. Is this constraint also true for usability?

Usability is not usually considered during the design of the software architecture due to the widespread assumption amongst software engineers that usability has to do only with the visible part of the system, the user interface. However, this fallacious dichotomy does not address all the usability concerns. Separation of the UI from the core functionality of the system seemed sufficient to support usability. Unfortunately, the modification of certain aesthetic aspects of the UI might require the rework of the software architecture. More specifically, certain usability driven modifications might incur modifications that cannot be easily accommodated by the architecture of the system.

Only two research groups have investigated the support for those aspects of usability that are connected to the design of the architecture. They suggested an approach to improve the usability of software systems by means of software architectural decisions by identifying specific connections between usability requirements and software architecture. These requirements are difficult to retrofit into a system if their absence has been discovered when the system is delivered to the customer. This leads us to conclude that more attention should be devoted to usability driven architectural analysis methods. We present a case study that consists in adapting an existing software architecture analysis method (SAAM) for the purpose of deriving

the interdependencies between architectural characteristics and usability requirements. More specifically, we investigate the impact of implementing usability requirement changes on the architecture. Potential design solutions that accommodate the corresponding usability mechanisms into the web software architecture are presented, along with their rational and the process by which they are obtained. After a detailed analysis of two distinct architectures of a web system implemented by software engineering students at Ecole Polytechnique de Montreal, our study shows that two of the six chosen scenarios cannot be implemented into an existing system without incurring major architectural changes. These impacts could have been avoided if the usability requirements were defined and considered in the architectural description phase. Both teams constructed their architectures without paying sufficient attention to usability requirements and whether it can easily accommodate late modifications.

Fixing usability problems on an already implemented system can prove costly if the changes require modifications that reach beyond the presentation layer, namely those that cannot be easily accommodated by the software architecture. Taking into account some usability requirements earlier in the software development cycle, more specifically before architectural design, can reduce the cost of these modifications. However, there is scarcity of methods and guidelines with the scope of directing users in eliciting the usability requirements that can impact a software architecture. We propose a usability driven adaptation of the quality attribute workshop (QAW) to assist software development organizations in discovering and documenting those usability requirements. An empirical experiment, conducted to assess the utility of this method, reveals that participants were successful in identifying the architecturally relevant usability requirements. It also helped discern the usability aspects that would not have been necessarily defined if this technique had not been employed. We show how this method could be integrated within an existing architecture-centric software development process.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

LISTE DES SIGLES ET ABBREVIATIONS

HCI :	Human Computer Interaction
IHM:	Interface Homme Machine
IU:	Interface Utilisateur
HTML:	Hyper Text Markup Language
J2EE :	Java Entreprise Edition
JSP:	Java Server Pages
QAW:	Quality Attribute Workshop
SAAM:	Software Architecture Analysis Method
SALUTA:	Scenario based Architecture Level UsabiliTy Analysis
STATUS:	SofTware Architecture That supports Usability
RUP:	Rational Unified Process
RUPi:	Rational Unified Process for Interactive Systems
UPEDU:	Unified Process for Education
UQAW:	Usability Quality Attribute Workshop

LISTE DES ANNEXES

INTRODUCTION

L'informatique prend une place grandissante dans notre vie quotidienne. La plupart des dispositifs de la maison (téléphone, répondeur, etc.) fonctionnent à l'aide d'un microprocesseur et d'un programme informatique. Ces outils, rendus « intelligents » par l'informatique, ont été conçus pour rendre service et nous faciliter la vie. Et effectivement, ils nous permettent de réaliser facilement certaines tâches qui, sans eux, auraient demandé beaucoup plus de temps et d'énergie. Mais qui n'a jamais éprouvé quelques difficultés à les utiliser? Qui n'a jamais ressenti ce sentiment de frustration de ne pas pouvoir utiliser pleinement un objet parce qu'il ne sait pas comment s'en servir? En fait, l'usage d'un instrument se caractérise par son utilisabilité[1] qui représente la capacité de l'objet à être facilement utilisé par une personne donnée pour réaliser la tâche pour laquelle il a été conçu. La notion de l'utilisabilité englobe à la fois la performance de réalisation de la tâche, la satisfaction que procure l'utilisation de l'objet et la facilité avec laquelle on apprend à s'en servir. Cette qualité, que l'on relie à l'ergonomie, concerne tout type d'instrument destiné à aider l'être humain dans son travail. Nous nous intéressons ici à son application au logiciel.

Lorsqu'un logiciel est employé à des fins professionnelles, son utilisabilité est essentielle car elle déterminera sa performance. Un logiciel facilement utilisable permettra de réaliser rapidement la tâche prévue, sans perte de temps et avec moins de stress. De ce fait, pour une entreprise, l'utilisabilité est un critère de choix important, qui conditionne la réussite commerciale d'un logiciel. N'est-ce pas la facilité d'utilisation qui a permis à Microsoft de tenir le haut du marché tandis qu'à l'inverse, Unix nous comblait par sa puissance et sa concision? Le succès commercial d'un produit informatique n'est donc pas uniquement lié à sa fonctionnalité. Le choix du consommateur se porte vers le logiciel le mieux adapté à son besoin et à ses compétences. En général, le client est prêt à faire des concessions en terme de

[1] « Utilisabilité » est une traduction littérale de *Usability*

fonctionnalité et de performance lorsqu'il sait l'outil est agréable à utiliser et qu'il ne perdra pas de temps à apprendre à s'en servir.

Finalement, lors de mise en œuvre des grands systèmes informatiques, l'utilisabilité est une condition majeure de réussite du projet. Un sondage (Tableau 1), mené auprès d'un échantillon de 8000 gestionnaires de petites, moyennes et grandes entreprises dans les principaux secteurs industriels montre que le facteur le plus important de l'échec de projets informatiques est le manque d'implication de l'utilisateur (Standish 1995). La facilité d'utilisation ou aussi la convivialité, sont des qualités importantes lorsque nous nous servons des logiciels. Elles vont nous permettre de nous concentrer sur la tâche proprement dite et non sur la manière de la réaliser. Lorsque le logiciel est simple à utiliser, nous nous intéressons au "quoi faire ? " et non à "comment faire ? ".

Dans le domaine des logiciels, la concurrence devient de plus en plus vive et les compagnies doivent concevoir les applications avec des interfaces graphiques de très haute qualité. Les utilisateurs ont tendance à choisir des logiciels avec une interface facile à utiliser. C'est pour cette raison qu'il faut offrir à l'utilisateur une interface qui soit à la fois plaisante et facile à utiliser, lui permettant de manipuler efficacement le contenu qui lui est destiné (Nielsen 1993). En effet, pour le client, la qualité de l'interface est un critère d'acceptation du produit final. Dorénavant, puisque la conception des interfaces est un processus complexe et itératif, des modèles de conception (aussi appelés des modèles d'architecture) ont été proposés pour assurer le déploiement de systèmes informatiques avec des interfaces ergonomiques. Ces modèles d'architectures reposent sur le principe de séparation du noyau fonctionnel de l'interface. Les apports d'une telle organisation modulaire sont importants. D'une part, cette organisation permet une conception plus simple dans la mesure où chaque module peut être réalisé de manière plus ou moins indépendante. D'autre part, les modifications apportées aux modules s'en trouvent amoindris et leur fiabilité accrue.

Désormais, de récentes recherches ont identifié des aspects d'utilisabilité qui ont des implications architecturales. Plus précisément, certaines modifications de l'interface

vont au-delà de la couche de la présentation et nécessitent une considération dans l'architecture logicielle. Nous avançons ces travaux en :

- Proposant une méthode pour évaluer l'utilisabilité tôt dans le cycle de développement, plus précisément son soutien dans l'architecture. Cette technique va aussi permettre le discernement de l'impact de changements d'utilisabilité sur une architecture existante.
- Proposant une méthodologie pour guider les intervenants dans l'identification des aspects d'utilisabilité qui ont un impact architectural, lors de l'analyse des exigences du client.

Le premier chapitre présente une revue de la littérature pertinente de l'utilisabilité tandis que le chapitre 2 expose le lien entre l'utilisabilité et l'architecture. Nos objectifs ainsi que notre motivation de recherche seront illustrés dans le chapitre 3. Le chapitre 4 reproduit le premier des deux articles présentés dans le cadre du mémoire, alors que le second article est présenté au chapitre 5. Finalement, nous présenterons une discussion globale ainsi que les limitations de nos recherches, complémentaires aux conclusions énoncées à l'intérieur des articles.

CHAPITRE 1

L'UTILISABILITÉ, UN IMPORTANT ATTRIBUT DE QUALITÉ LOGICIELLE

1.1 Une vue multidimensionnelle de la qualité logicielle

Depuis quelques années, la qualité des systèmes informatiques est devenue de plus en plus importante dans la vie de tous les jours. Chacun de nous est en contact en quelque sorte avec le logiciel. On le trouve dans les véhicules à moteur, les avions, les usines chimiques ou nucléaires, les feux de signalisation, les diagnostiques médicaux, les tests d'équipement et les systèmes de communication, pour en nommer quelques-uns. Il est clair qu'un logiciel de mauvaise qualité pourrait causer de graves dommages et les ingénieurs responsables de la qualité logicielle doivent continuellement trouver des façons de mettre en valeur la sûreté de fonctionnement de ces systèmes. Malheureusement, la qualité est en général, une notion ambiguë : cette ambiguïté réside dans la multitude de ses définitions dans chaque domaine, la rendant multidimensionnelle. Les différentes dimensions sont résumées par Garvin (1984) en cinq vues:

- *La vue transcendantale* : la qualité est quelque chose qu'on peut reconnaître, mais qu'on ne peut pas définir.
- *La vue utilisateur* : la qualité est la force apparente du produit pour réaliser des fonctions.
- *La vue de fabrication* : la qualité est la conformité aux spécifications.
- *La vue du produit* : la qualité est attachée aux caractéristiques intrinsèques du produit. C'est la vue la plus évoquée par les spécialistes de la qualité du logiciel.
- *La vue basée sur la valeur* : la qualité dépend des coûts du produit.

En effet, Kitchenham et Pfleeger (1986) suggèrent que la qualité logicielle, en prenant comme référence sa nature multidimensionnelle, peut principalement être considérée en trois facettes :

- Qualité du produit (point de vue fonctionnel): la vue du produit
- Qualité du produit (point de vue utilisation): la vue utilisateur
- Qualité du processus: la vue de fabrication

1.2 La qualité du produit : un point de vue fonctionnel

Si on demande à différentes personnes impliquées dans le développement d'un logiciel d'indiquer les caractéristiques de qualité externes (utilisabilité) et internes (fiabilité, maniabilité, etc.) qu'elles jugent importantes, il est fort probable que leurs réponses seront différentes. Cela peut être expliqué par le fait que l'importance de ces caractéristiques dépend du profil de la personne qui analyse le produit. Par exemple, un produit peut être analysé par des utilisateurs afin de vérifier sa facilité d'utilisation, mais aussi par les concepteurs qui doivent s'assurer que le système est assez robuste et stable et que sa maintenance ne va pas encourir de sérieux changements plus tard. Une définition claire et juste de la qualité demeure encore un souci pour la communauté du génie logiciel. Des modèles et des standards de qualité doivent être utilisés pour identifier les caractéristiques externes et internes qui peuvent représenter la vue de l'utilisateur et du concepteur.

Au début des années 80, la communauté de la qualité du logiciel en regroupant les différentes vues de la qualité (par exemple celle de : McCall et Walters 1977; Boehm, Brown et Kaspar 1978), créa un modèle de qualité qui ensuite deviendra un standard universel. Le fruit de cette tentative a été le standard ISO 9126 (2003) (Figure 1.1). Cette organisation hiérarchique des différentes caractéristiques de la qualité d'un produit est composée de six facteurs de qualité : *la fonctionnalité*, *la fiabilité*, *l'efficacité*, *la facilité d'utilisation*, *la maniabilité* et *la portabilité*. Le standard prétend que ces six facteurs couvrent tous les aspects de la qualité.

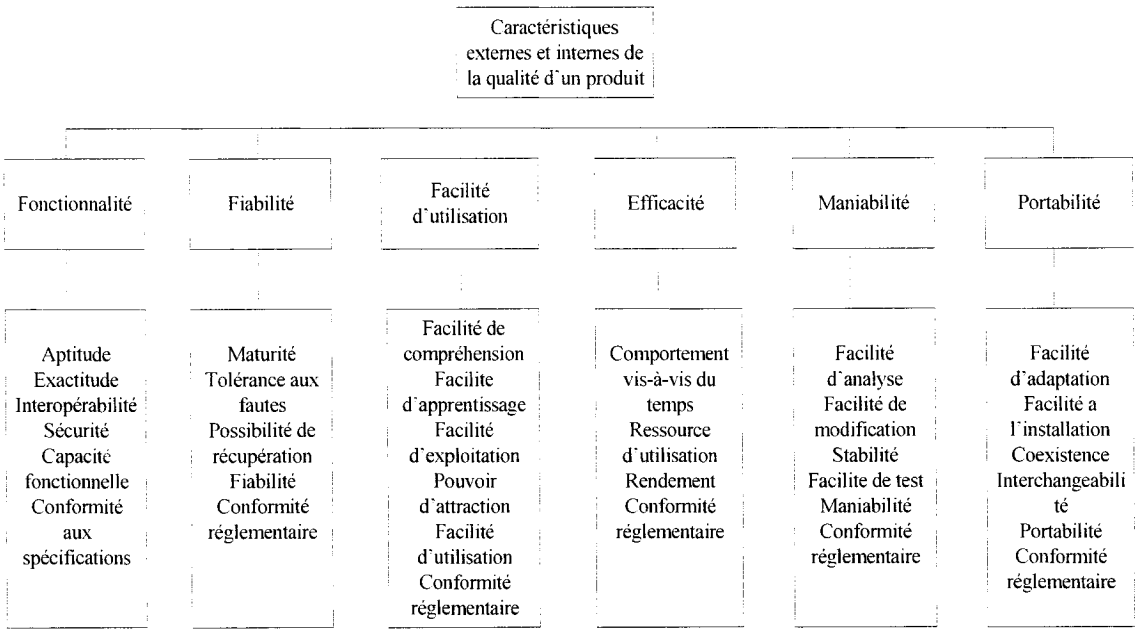


Figure 1.1: Le modèle de qualité du logiciel de la norme ISO 9126

1.3 La qualité du produit : un point de vue utilisation

Les années 1980 verront naître les premiers essais de définition de la notion d'utilisabilité, notamment dans les travaux de Shackel (1981, 1986), Eason (1984), et se développera graduellement l'idée d'une *conception centrée utilisateur* (Norman 1986, Karat et Bennett 1991). Jakob Nielsen (1993), un gourou de l'utilisabilité l'a défini comme suit :

> *Un attribut de qualité qui permet d'évaluer la facilité d'utilisation des interfaces utilisateurs. Le terme "utilisabilité" se réfère également aux méthodes permettant d'améliorer l'aisance d'utilisation au cours du processus de conception.*

Dans un des derniers raffinements de la définition, Shackel (1991) décrit l'utilisabilité d'un système comme :

> *Sa capacité, en termes fonctionnels humains, à permettre une utilisation facile et effective par une catégorie donnée d'utilisateurs, avec une formation et un support adapté, pour accomplir une catégorie donnée de tâches.*

Cette définition met l'accent sur des mesures classiques de la performance en psychologie cognitive : vitesse d'accomplissement de la tâche et taux d'erreurs. Il est encourageant de constater que la définition donnée par Shackel (1991) est très proche de celle qui a été inclus dans la norme ISO 9241-11 (1998) de l'Organisation Internationale de Normalisation, que l'on peut considérer comme une définition consensuelle :

> *L'utilisabilité est le degré selon lequel un produit peut être utilisé, par des utilisateurs identifiés, pour atteindre des buts définis avec efficacité, efficience et satisfaction, dans un contexte d'utilisation précis.*

Cette définition indique que l'utilisabilité est la conjonction de trois éléments que sont l'efficacité, l'efficience et la satisfaction. Ces trois éléments sont développés dans les sections suivantes.

1.3.1 L'efficacité

L'efficacité se réfère à la capacité d'un utilisateur à atteindre un objectif donné. La norme ISO 9241 l'a défini comme la précision ou degré d'achèvement selon lesquels l'utilisateur atteint des objectifs spécifiés, se concentrant ainsi sur la mesure du résultat obtenu, la performance. L'évaluation de l'efficacité suppose d'avoir défini préalablement, d'une manière plus ou moins contraignante, les objectifs à atteindre. Classiquement, deux mesures associées à l'efficacité à atteindre sont distinguées :

- La *réussite de la tâche* : la capacité à atteindre minimalement, partiellement ou totalement les objectifs fixés;
- La *qualité de la performance* : la capacité à atteindre les objectifs avec le plus de précision possible.

1.3.2 L'efficience

L'efficience est la capacité de produire une tâche donnée avec le minimum d'effort - plus l'effort est faible, plus l'efficience est élevée. En termes ergonomiques, cette notion renvoie, par exemple, à l'évaluation de la charge de travail (physique et mentale) imposée à l'utilisateur. La norme ISO définit l'efficience comme étant le «rapport entre les ressources dépensées et la précision et le degré d'achèvement selon lequel l'utilisateur atteint des objectifs spécifiés». Replacée dans un contexte d'utilisation, parfois les utilisateurs préfèrent dépenser moins de ressources physiques ou cognitives et se contentent d'un résultat moins satisfaisant. Globalement, cinq types d'indicateurs peuvent être pris en compte dans l'évaluation de l'efficience d'un système informatique, à savoir :

- Le taux et la nature des erreurs d'utilisation; (plus une erreur est irréversible, plus elle occasionnera une baisse de l'efficience)
- Le temps pour exécuter une tâche donnée;
- Le nombre d'opérations requises pour exécuter une tâche principale;
- La charge de travail (se réfère au coût cognitif de la réalisation d'une tâche)
- Facilité d'apprentissage (le temps d'apprentissage du fonctionnement d'un site doit être proche de zéro pour tout visiteur, tant celui-ci aime à trouver facilement ce qu'il est venu chercher).

1.3.3 La satisfaction

La satisfaction se réfère au niveau de confort ressenti par l'utilisateur lorsqu'il utilise un produit. La satisfaction correspond à une réaction affective qui concerne l'acte d'usage d'une application et qui peut être associée au plaisir que l'utilisateur ressent en échange de son acte. La satisfaction est une évaluation subjective provenant d'une comparaison entre ce que l'acte d'usage apporte à l'individu et ce qu'il s'attend à recevoir. Cependant, cet indicateur est difficile à mesurer de par sa nature subjective.

Dans la plupart des cas, des échelles d'évaluation dites subjectives se sont imposées. A travers un questionnaire de satisfaction, l'utilisateur exprime son sentiment global sur un certain nombre d'aspects liés à l'interaction homme-machine.

Les différents composants de l'utilisabilité – efficacité, efficience et satisfaction – ne sont pas isolés les uns des autres. En effet, plusieurs recherches ont établi des corrélations entre eux, et ces relations peuvent s'énoncer ainsi :

- Un système efficient et facile à apprendre est nécessairement efficace mais le contraire n'est pas inévitablement vrai.
- Un système efficace favorise mais n'aboutit pas obligatoirement à un sentiment de satisfaction. La satisfaction étant une variable complexe, l'utilisateur peut être insatisfait pour des raisons diverses même si le logiciel lui permet d'accomplir la tâche voulue.
- Finalement, un utilisateur peut être satisfait d'un dispositif qui n'est pas forcément efficient. La satisfaction est due à des raisons psychosociologiques qui peuvent être liés à des effets de mode.

1.4 La qualité du processus : un point de vue fabrication

1.4.1 Intégration des facteurs humains dans le développement

Les utilisateurs finaux sont les mieux placés pour influencer le développement d'un produit. Si le produit final correspond à leurs besoins et caractéristiques, il aura toute la chance d'être adopté. Et c'est bien sûr le but ultime de tout produit. Malheureusement, les processus de génie logiciel traditionnels sont tout à fait adéquats pour le déploiement de systèmes informatiques fonctionnels, mais ne sont adaptés pour la conception de systèmes utilisables. La plupart des auteurs s'entendent sur la conception des systèmes interactifs centrés sur l'utilisateur comme un élément primordial pour garantir l'utilisabilité de l'interface (Bastien et Scapin, 2003; Seffah, Gulliksen et Desmarais

2005). De plus, Scapin (1986) identifie plusieurs défauts dans les cycles de développement qui ne sont pas centrés sur l'utilisateur. Ces défauts dans le cycle de développement peuvent rendre le logiciel inutilisable. Par exemple, lorsque l'utilisateur n'intervient pas au niveau de la conception et du développement, le logiciel développé ne répondra pas à ses besoins:

- Manque de connaissances préalables des tâches et des utilisateurs.
- Utilisation de méthodes informatiques fonctionnelles n'incluant pas la prise en compte de l'opérateur humain.
- Logique de fonctionnement plutôt qu'une logique d'opération.
- Manque d'homogénéité dans la conception.
- Ne pas prévoir les erreurs humaines.
- Conception selon des critères de performance des systèmes plutôt que des critères liés aux objectifs des utilisateurs et aux contraintes de la tâche.
- Fournir toutes les fonctions imaginables plutôt qu'un ensemble plus réduit de fonctions essentielles.
- Fournir toutes les informations disponibles plutôt que seulement celles nécessaires à la tâche.

Ce cycle itératif de conception comporte une phase de planification du processus centré sur l'utilisateur, une phase d'analyse des besoins utilisateurs et organisationnels, une phase de conception, et une phase d'évaluation. Ces phases se décomposent en activités qui sont, par exemple : l'identification des profils des utilisateurs, l'analyse de l'activité des tâches, l'établissement des objectifs d'utilisabilité, le prototypage, la conception détaillée, l'implantation et la rétroaction des utilisateurs.

La conception centrée utilisateur impose alors que le développement du produit soit guidé par les besoins des utilisateurs plutôt que par les responsabilités et les exigences techniques. Cette implication des utilisateurs doit être à la fois précoce (elle est nécessaire dès les prémisses du projet) et itérative (elle doit se répéter tout au long des

étapes clés du projet). Concevoir une application facile à utiliser est donc un résultat qui découle de méthodologies de conception, et nécessite de se demander à chaque étape critique de la conception si le produit correspond aux besoins des utilisateurs finaux.

1.4.2 ISO 13407 : Processus de conception de systèmes interactifs centrés sur l'humain

Cette approche a été traduite en une norme internationale, l'ISO 13407 (Processus de conception de systèmes interactifs centrés sur l'humain). Cette norme définit les conditions de la mise en œuvre d'un processus centré sur l'opérateur humain. Cinq principes sont nécessaires à la satisfaction de cette norme :

- *Une préoccupation en amont des utilisateurs*, de leurs tâches et de leur environnement.
- *La participation active de ces utilisateurs*, ainsi que la compréhension claire de leurs besoins et des exigences liées à leurs tâches.
- *Une répartition appropriée des fonctions.*
- *L'itération des solutions de conception.*
- *L'intervention d'une équipe de conception multidisciplinaire.*

De façon plus précise, l'ISO 13407 (Figure 1.2) définit les étapes du cycle de conception centrée utilisateur (CCU) comme suit:

- *Planification du processus*: L'équipe projet doit avoir atteint un consensus concernant la recherche et la satisfaction de la norme ISO 13407 et donc de ses implications sur les plans techniques, méthodologiques, et de conduite de projet. De façon résumée, cette pré-étape consiste à planifier les activités de développement dans une optique de conception centrée utilisateur.
- *Comprendre et spécifier le contexte d'utilisation*: La première étape proprement dite du cycle de CCU vise à comprendre et spécifier le contexte d'utilisation. Il s'agit donc de comprendre la population cible et ses caractéristiques, ses buts et

tâches, et ses environnements. L'identification des profils d'utilisateurs est la base essentielle de cette étape. La connaissance de ces profils (compétences, fonctions, tâches à accomplir, niveau d'expérience, éducation, etc.) permettra de choisir les méthodes d'évaluation appropriées et de sélectionner des participants pour mener des tests utilisateurs.

- *Comprendre et spécifier les exigences*: Cette étape est concernée par l'identification des besoins, compétences et environnement de travail de tous les intervenants pertinents au système. Les documents de spécifications consistent en des descriptions précises des profils d'utilisateurs et des cas d'utilisation. Les exigences sont ensuite ordonnées selon leur importance. A ce niveau, les objectifs d'utilisabilité vont guider et justifier les choix de conception. Ils fourniront par la suite des critères d'acceptation lors des tests utilisateurs.
- *Produire des solutions de conception*: L'étape de production de solutions de conception vise à utiliser les connaissances acquises lors des étapes précédentes pour matérialiser les solutions afin de pouvoir les modifier en fonction des rétroactions utilisateurs.
- *Évaluer les solutions*: Les prototypes créés au stade précédent sont utilisés pour évaluer les solutions conçues en fonction des exigences. L'objectif de cette phase est de recueillir une rétroaction sur la conception développée. C'est principalement une évaluation de la satisfaction des objectifs utilisateurs et organisationnels.

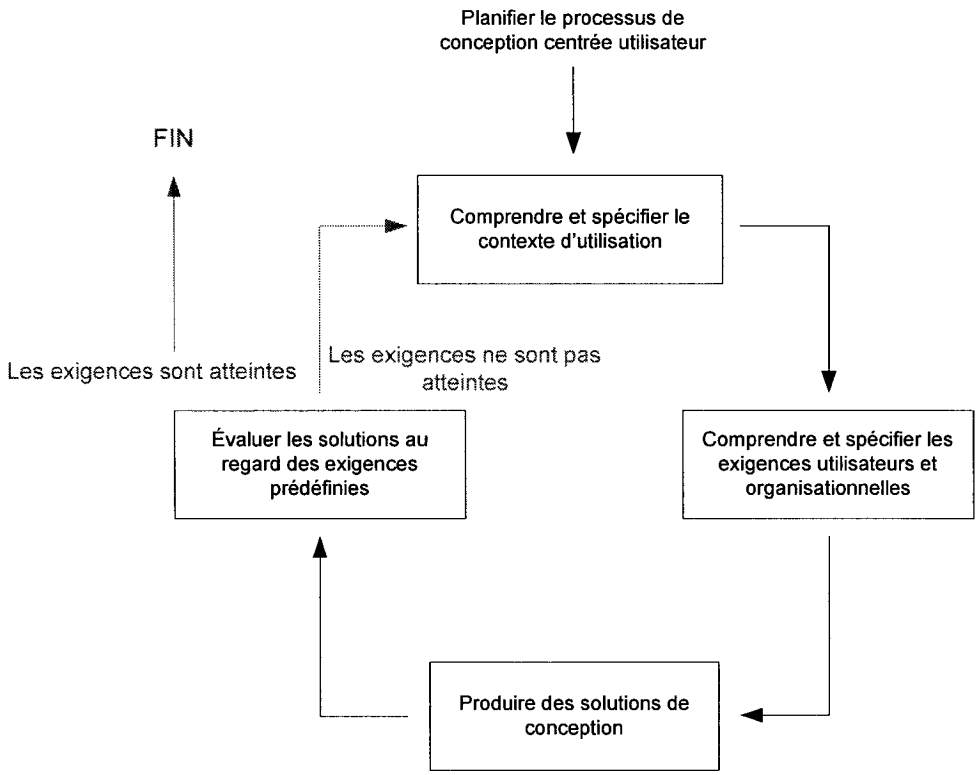


Figure 1.2: Processus ISO 13407

1.5 L'aspect graphique de l'utilisabilité

1.5.1 L'interface graphique utilisateur

La conception de l'interface est l'élément déterminant pour l'utilisabilité d'un système. L'interface est, comme le définit Rhéaume (1999), la couche la plus proche de l'utilisateur et elle joue un grand rôle dans l'utilisabilité d'un système informatique. L'interface occupe une part importante du code développé, en moyenne 48%, voire 80% pour les applications Web (Schneiderman 1998). Peu importe les nombreuses caractéristiques qui peuvent différencier les utilisateurs (motivation, aptitude, habiletés, etc.), tout système qui offre une interface qui n'est pas d'un grand soutien ne peut pas prétendre être d'une grande valeur.

1.5.2 Test d'utilisabilité : évaluation de l'interface

Une étape importante après la conception de l'interface est l'évaluation ergonomique. Ces évaluations permettent de détecter les problèmes liés à l'utilisabilité (Scapin 1986, Bastien 1991, Dumas et Redish 1993, Rubin 1994, Nielsen 2002). La classification la plus fréquente est fondée sur la distinction entre les approches prédictives et approches expérimentales (Figure 1.3). Par contre, Senach (1990) parle respectivement de méthodes analytiques et de méthodes empiriques. Les paragraphes qui suivent illustrent ces points de vue.

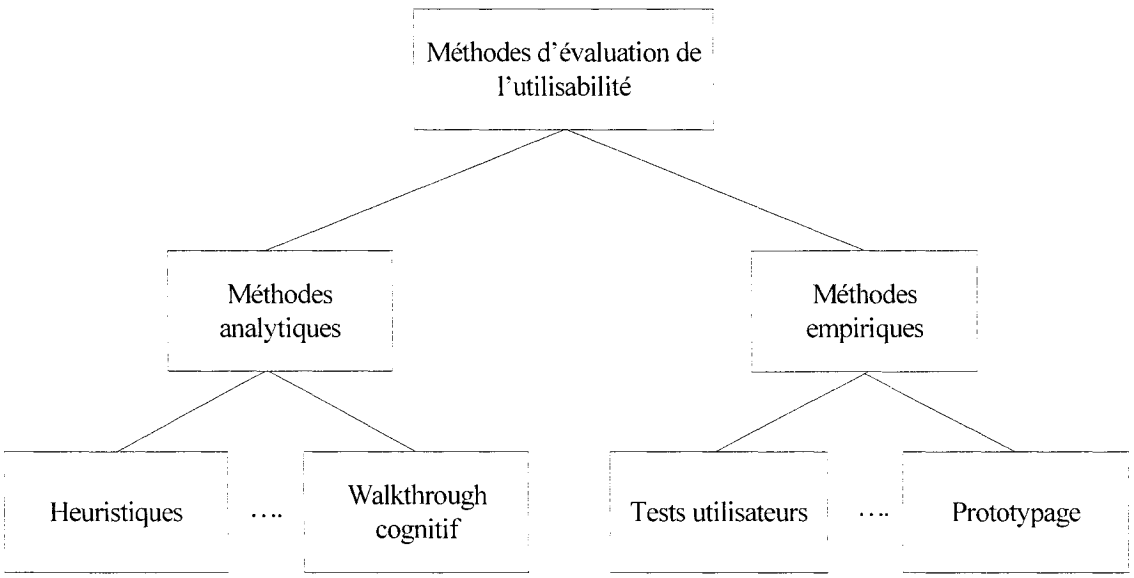


Figure 1.3: Classification des méthodes d'évaluation de l'utilisabilité

1.5.2.1 Les méthodes analytiques

Les méthodes analytiques permettent, à partir d'une description du système, d'identifier les problèmes potentiels d'utilisabilité. Les approches heuristiques aussi bien le *walkthrough cognitif* s'appuient sur la connaissance d'évaluateurs experts qui détectent des difficultés potentielles.

L'évaluation heuristique est une méthode d'inspection de l'interface homme-machine appuyée sur un ensemble de critères ergonomiques (Smith 1986) aussi bien

que sur des règles simplifiées comme celles de Scapin (1986) et Nielsen (1992). Elle permet d'identifier des lacunes qui peuvent ainsi être corrigées avant que le produit n'ait été complètement développé. Cette technique est relativement facile à utiliser une fois qu'on en maîtrise les éléments de base. Il faut toutefois se rappeler que l'évaluation heuristique n'est pas une panacée et qu'elle ne saurait remplacer une démarche de conception appropriée.

Le *walkthrough cognitif* est une procédure spécifique pour simuler les processus cognitifs des utilisateurs quand ils sont en interaction avec une interface. La méthode nécessite une description minimale du prototype d'interface du système, une description de la tâche que l'utilisateur effectuera, une liste des actions et une description de l'usager.

1.5.2.2 Les méthodes empiriques

A l'inverse des méthodes analytiques, les approches empiriques, aussi appelées expérimentales, se basent sur des données recueillies par des observations, des questionnaires et des tests utilisateurs. Elles nécessitent le recrutement des utilisateurs cibles afin de réaliser des tâches typiques de l'utilisation du système.

Les tests utilisateurs permettent d'évaluer l'interface dans le but de mesurer empiriquement certains aspects d'utilisabilité tout en observant le comportement des utilisateurs face à l'application. Ces tests permettent de mesurer le taux de réussite d'une tâche, le nombre d'erreurs et le taux d'apprentissage. Nielsen et Landauer (1993) ont démontré que le nombre de problèmes d'utilisabilité découverts avec n utilisateurs peut s'exprimer de la manière suivante :

$$N(1-(1-L)^n)$$

Où N dénote le nombre total de problèmes d'utilisabilité et L représente une constante liée à la proportion de problèmes trouvés par un seul utilisateur.

CHAPITRE 2

L'UTILISABILITÉ ET LE GÉNIE LOGICIEL

2.1 Intégration de l'utilisabilité dans les processus de génie logiciel

2.1.1 Problématique

Dans le cadre d'un projet de développement logiciel, la pratique de l'utilisabilité permet de réduire les coûts. En effet, comme c'est prescrit dans la norme ISO 13407, en impliquant l'utilisateur dès la phase de conception, l'équipe technique peut rapidement consolider avec le client sa compréhension des requis. Une meilleure connaissance de l'utilisation effective du logiciel et de l'attente des utilisateurs permet d'éviter les retours en arrière, donc les surcoûts, dus à une mauvaise compréhension des requis.

Bien entendu, l'utilisation de cette approche de conception centrée utilisateur (c'est-à-dire en impliquant l'utilisateur tout au long du développement) a quand même des inconvénients. Gulliksen et Lantz (1998) ont observé plusieurs obstacles à franchir:

- *Problèmes de compétences et d'outils*: L'équipe de développement n'est pas tout à fait familière avec ce processus et la plupart de leurs outils ne sont pas adaptés à ce type de méthodologie.
- *Manque de support*: Il doit y avoir un soutien de la direction et des utilisateurs. Le temps et les ressources nécessaires doivent être conformément alloués à un projet pour que les utilisateurs puissent participer le plus possible.
- *Aspects externes*: Certains aspects inattendus peuvent troubler la conception centrée utilisateur. Différents conflits d'intérêts qui peuvent apparaître exigeront une intervention externe.

Pour remédier à ces problèmes, de solides liaisons de collaboration et de communication entre les processus de génie logiciel et les meilleures pratiques de la conception centrée utilisateur doivent être établies, puisqu'ils partagent les mêmes objectifs (c'est-à-dire le déploiement de systèmes informatiques que les utilisateurs pourront utiliser afin d'accomplir leur travail sans compromettre leur performance). Dans les systèmes informatiques d'aujourd'hui, 37 à 50 % des efforts produits durant le cycle de développement de l'application sont liés à l'interface utilisateur (Myers et Rosson 1992). Pour cette raison, les méthodes et pratiques du domaine de l'Interaction Homme Machine (IHM) affectent le processus de génie logiciel. Malheureusement, aucune coordination entre ces deux méthodologies existe puisqu'elles sont, dans la plupart du temps, appliquées indépendamment pendant le développement. Les experts d'utilisabilité sont fréquemment introduits dans le processus de développement pendant ou même après l'implantation pour soigner les problèmes d'utilisabilité d'un système déjà existant. Et en raison de contraintes de temps très restreintes pour livrer le logiciel, les changements suggérés par ces experts sont souvent nombreux mais ceux que les ingénieurs logiciels acceptent d'implanter sont désormais de nature cosmétique, uniquement liés à l'interface. De ce fait, ces conflits aboutissent à des systèmes qui manquent non seulement d'utilisabilité mais aussi de fonctionnalité. Clairement, la synchronisation des meilleures pratiques d'utilisabilité avec les processus de génie logiciel est indispensable pour développer une application qui répond à la fois aux exigences fonctionnelles du client et qui sera utilisable (Seffah, Gulliksen et Desmarais 2005). L'accent devrait être porté vers l'affectation de certaines techniques spécifiques à l'utilisabilité que les ingénieurs logiciels pourront utiliser et incorporer dans leurs processus déjà existants. Les sections qui suivent illustrent certaines des récentes recherches qui ont émergées.

2.1.2 Les travaux de Krutchen

Krutchen et ses collègues (2001) décident d'inclure le rôle de concepteur IU responsable de la modélisation de l'IU et du prototypage de l'IU dans la discipline de définitions des besoins du RUP. Au cours de l'activité de modélisation, le concepteur IU analyse le modèle de cas d'utilisation et crée un scénarimage de cas d'utilisation. Cet artefact se compose d'une description textuelle et l'interaction homme-système, des diagrammes de classes, des références au prototype de l'IU. Le concepteur IU va ensuite procéder à la conception et l'implantation du prototype tout en obtenant la rétroaction des autres membres de l'équipe de développement et des utilisateurs.

2.1.3 Les travaux de Ferre

Ferre (2003) aborde l'intégration des activités d'utilisabilité dans un processus de développement générique en utilisant la terminologie et les concepts que les ingénieurs logiciels pourront assimiler. Il examine dans un premier volet la littérature pour identifier les caractéristiques qui définissent un processus centré utilisateur et choisit par la suite les techniques et les activités les mieux adaptées à être incluses dans un processus logiciel. Ensuite, il associe ces techniques à leurs activités respectives dans un processus pour enfin les distiller en tant qu'incréments qui doivent être incorporés à différentes phases du processus.

2.1.4 Les travaux de Sousa et Furtado

Sousa et Furtado (2003b) se concentrent sur l'adaptation du *Rational Unified Process*® (RUP) (Krutchen 2001) pour y intégrer des aspects d'utilisabilité dans ses disciplines. Elles proposent le *Rational Unified Process for Interactive Systems* (RUPi) qui prend en considération l'utilisateur et son contexte d'utilisation, ainsi que les mécanismes pour concevoir et tester les interfaces graphiques. Elles ont modifié certaines disciplines existantes du RUP en ajoutant des nouvelles activités et de

nouveaux rôles et artefacts qui seront produits par ces derniers. Chaque discipline suggérera la génération d'artefacts qui est censée aider l'équipe de développement dans la conception de systèmes interactifs en considérant principalement les facteurs humains, la conception de l'interface et les tests d'utilisabilité.

2.1.5 Les travaux de John et Bass

John et ses collègues de l'institut de génie logiciel (SEI) (2003) ont examiné la norme ISO 13407 ainsi que le processus RUP recherchant les points de communication qui sont explicitement mentionnés aussi bien que les asymétries. Ils ont principalement comparé *ISO 13407 : Human-centred design processes for interactive systems* (1999) et le *Rational Unified Process : Best practices for software development teams* (2001). Leurs résultats sont présentés dans le tableau 2.1.

Tableau 2.1: RUP versus ISO 13407 - similitudes et asymétries

Similitudes	Asymétries
• Les deux processus perçoivent la phase de planification de projets comme une étape importante au succès du projet. • La définition et la spécification des requis utilisateurs sont indispensables. • Les deux processus sont considérés itératif. Dorénavant, ils n'ont pas une définition commune pour itérations. • Les deux processus abordent la phase de test comme un critère important dans l'acceptation du système. De plus, ils n'effectuent pas les tests de la même manière.	• ISO 13407 exige que du personnel compétent soit embauché et que des méthodes appropriées soient utilisées, alors que le RUP ne fait aucun contrôle sur l'équipe de développement ni sur les méthodes, supposant que tous les membres sont qualifiés pour effectuer leur travail. • RUP exige une première étude des objectifs d'affaires et une analyse des risques tandis que ISO 13407 ne mentionne pas de tels documents. • RUP met le point sur l'établissement de l'architecture d'une application dans la phase d'élaboration alors qu'ISO 13407 passe directement du prototype à la conception détaillée.

2.2 L'utilisabilité : vers une nouvelle perspective

L'interface utilisateur doit satisfaire à des critères d'utilisabilité, c'est-à-dire à la fois permettre aux utilisateurs d'atteindre leurs objectifs à travers des interactions intuitives et sûres mais aussi fournir une adéquation avec les besoins et les capacités de ceux-ci. Dans le but d'assurer ces critères, l'interface doit :

- Refléter exactement le comportement du système : c'est l'aspect génie logiciel
- Faciliter son utilisation : c'est l'aspect ergonomique
- Être accessible au plus grand nombre d'utilisateurs ayant des comportements différents : c'est l'aspect psychologique

Le domaine du IHM est maintenant bien plus qu'une préoccupation de l'utilisabilité de l'interface, et le recours à des modèles et à des méthodes s'avère indispensable afin d'assurer l'utilisabilité des systèmes interactifs. Ces modèles proviennent des différentes communautés (génie logiciel, ergonomie cognitive) et constituent un champ de recherche très actif. On s'intéresse ici aux modèles de conception issus du domaine du génie logiciel, appelés également modèles d'architecture.

2.3 Les architectures des applications interactives

Les modèles d'architecture sont nés du constat que la conception des interfaces est un processus complexe et itératif, et par conséquent la possibilité d'apporter des modifications aux logiciels à n'importe quel moment du cycle de développement doit être présente. Dans ce sens, la décomposition de l'application interactive en différents éléments est rendue nécessaire. Les modèles d'architecture définis pour le IHM reposent sur un principe commun : la séparation du modèle sémantique de l'application (noyau de l'application) de l'interface. Les modèles d'architecture décrivent également la manière dont ces deux composants communiquent entre eux. Ainsi, ils offrent une structure générique destinée au concepteur de l'application. D'une part, cette

organisation permet une conception plus simple dans la mesure ou chaque module peut être réalise de manière plus ou moins indépendante. D'autre part, les modifications apportées aux modules s'en trouvent amoindris et leur fiabilité accrue.

La section qui suit a pour objectif de présenter un survol des modèles d'architecture existants et de leur mise en œuvre. Il existe une abondante bibliographie sur ces sujets, dont on mentionne quelques références importantes

2.3.1 Modèle d'architecture en niveaux d'abstraction

Le modèle de Seeheim (Pfaff 1985) préconise la décomposition d'une application interactive en trois composants logiques : la présentation, l'interface avec le noyau fonctionnel et le contrôleur du dialogue (Figure 2.1)

- Le composant présentation se charge de l'affichage de l'information à l'utilisateur. Il interprète les actions de l'utilisateur par l'intermédiaire de la souris ou du clavier par exemple.
- Le composant interface avec le noyau fonctionnel convertit les entrées de l'utilisateur en appels du noyau fonctionnel et réciproquement traduit les données fournies par le noyau fonctionnel en représentations structurées présentables à l'utilisateur.
- Le composant contrôleur du dialogue joue le rôle de médiateur entre la partie présentation et l'interface avec le noyau fonctionnel. Il autorise et contrôle l'enchaînement des interactions de l'utilisateur

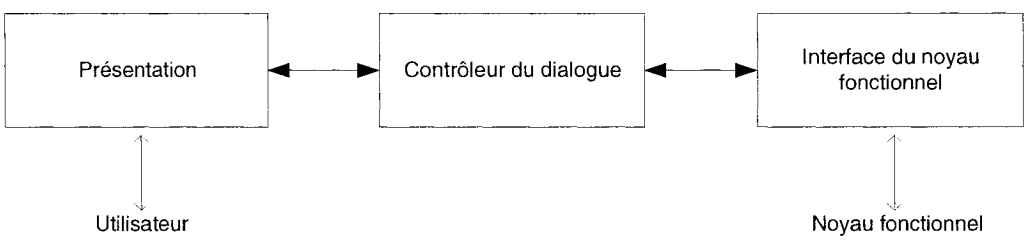


Figure 2.1: Le modèle Seeheim

Les avantages de cette approche incluent :

- Les problèmes de l'application et de l'interface peuvent être isolés et traités séparément.
- L'application et l'interface peuvent évoluer indépendamment.

2.3.2 Modèles d'architecture à agents

En décomposant les interfaces en objets de même nature, les modèles à agents sont proches des langages de programmation orientés objet. Dans cette section, nous décrivons les principaux modèles à agents, à savoir MVC et PAC.

2.3.2.1 Modèle–Vue–Contrôleur (MVC)

Le modèle MVC (Schmucker 1986; Krasner et Pope 1988) a été introduit comme le modèle d'architecture de référence dans l'implémentation des interfaces utilisateur de l'environnement Smalltalk (Goldberg et Robson 1983). Ce modèle d'architecture (Figure 2.2) impose la séparation entre les données, les traitements et la présentation, ce qui donne trois parties fondamentales dans l'application finale:

- Le *Modèle* représente le comportement de l'application : traitements des données, interactions avec la base de données. Il décrit les données manipulées par l'application et définit les méthodes d'accès.
- la *Vue* correspond à l'interface avec laquelle l'utilisateur interagit. Les résultats renvoyés par le modèle sont dénués de toute présentation mais sont présentés par les vues. Plusieurs vues peuvent afficher les informations d'un même modèle. La vue n'effectue aucun traitement, elle se contente d'afficher les résultats des traitements effectués par le modèle, et de permettre à l'utilisateur d'interagir avec elles.
- le *Contrôleur* prend en charge la gestion des événements de synchronisation pour mettre à jour la vue ou le modèle. Il n'effectue aucun traitement, ne modifie

aucune donnée, il analyse seulement la requête du client et se contente d'appeler le modèle adéquat et de renvoyer la vue correspondant à la demande

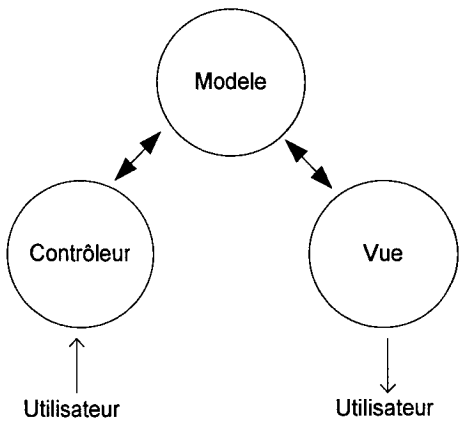


Figure 2.2: Le modèle MVC

En résumé, lorsqu'un client envoie une requête à l'application, celle-ci est analysée par le contrôleur, qui demande au modèle approprié d'effectuer les traitements, puis renvoie la vue adaptée au navigateur, si le modèle ne l'a pas déjà fait. Un avantage apporté par ce modèle est la clarté de l'architecture qu'il impose. Cela simplifie la tâche du développeur qui tenterait d'effectuer une maintenance ou une amélioration sur le projet. En effet, la modification des traitements ne change en rien la vue.

2.3.2.2 Présentation–Abstraction–Contrôle (PAC)

Le modèle à agents PAC (Coutaz 1987) réintroduit la vue et le contrôleur dans un objet monolithique mais rend explicite la synchronisation du modèle et de la vue/contrôleur. Ce modèle propose en outre une méthode de description récursive qui étend le paradigme à agents avec la notion de couche d'abstraction. Comme MVC, le modèle PAC décrit les systèmes interactifs comme une hiérarchie d'agents composée de trois modules.

- La *Présentation* gère l'interaction avec l'utilisateur.

- L'*Abstraction* encapsule la partie sémantique de l'agent et prend en charge les fonctionnalités de l'application.
- Le *Contrôle* maintient la consistance entre la présentation et l'abstraction, et permet aux deux composants préalablement cités de dialoguer.

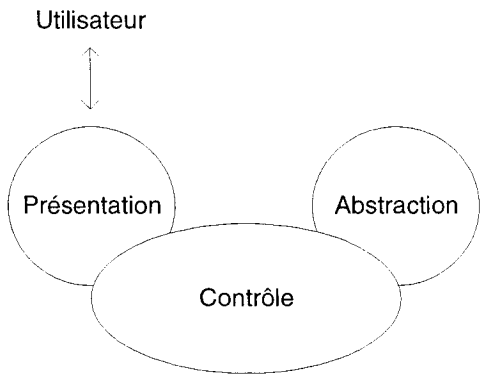


Figure 2.3: Le modèle PAC

Notons qu'ici le composant d'abstraction est l'équivalent du composant *Modèle* de MVC, et que la présentation correspond à une fusion des composants *Vue* et *Contrôleur*. Le composant *Contrôle* n'a pas d'existence explicite dans le modèle MVC.

2.3.3 Problèmes observés dans ces modèles d'architecture

Jusqu'à ce point, l'utilisabilité a été principalement identifiée comme étant une propriété de présentation de l'information. En fait, la séparation de l'interface du noyau de l'application (à l'aide des modèles d'architectures préalablement cités) rend la modification de l'interface plus facile après les tests d'acceptation. Cette dichotomie, d'une interface utilisateur séparée de sa fonctionnalité, ne tient pas compte du rapport intime qui existe entre les attributs internes d'un système logiciel qui peuvent affecter son utilisabilité.

De nos jours, beaucoup de logiciels souffrent de graves problèmes d'utilisabilité qui ne peuvent pas être corrigés sans encourir de sérieuses modifications à l'architecture. En fait, même si la présentation d'un système est bien conçue et que ces modèles d'architectures sont utilisés lors de la conception, l'utilisabilité d'une application peut être considérablement compromise si l'architecture n'accommode pas certains aspects d'utilisabilité. Il s'avère très difficile de faire des changements à un système afin d'améliorer son utilisabilité puisque plusieurs de ces changements exigent des modifications au niveau de l'application et ne pourront pas être facilement accommodés par l'architecture. Cependant ce n'est pas toujours le cas. Une étude effectuée sur logiciel de traitement d'images à source ouverte (GIMP) permet de conclure que certaines architectures peuvent en effet accommoder des modifications d'utilisabilité (Rafla, Oketokoun, Wiklik, Desmarais et Robillard 2004).

Après avoir entrepris une étude approfondie de la littérature de l'utilisabilité, très peu d'auteurs ont examiné comment l'utilisabilité peut être atteinte pendant la construction de l'architecture. Len Bass et ses collègues (2001; 2003), et le projet STATUS (**S**of**T**ware **A**rchitecture **T**hat supports **US**ability) (2002) ont chacun présenté une approche pour élucider le rapport entre l'utilisabilité et l'architecture de logiciel. Les sections qui suivent illustrent ces deux approches.

2.4 Le soutien architectural de l'utilisabilité

2.4.1 Le projet STATUS

Les recherches du STATUS (Folmer, Van Gurp et Bosch 2003a) sur le lien entre l'utilisabilité à l'architecture, ont résulté en une définition d'une méthodologie qui se compose de trois couches (Figure 2.4). Ils ont entrepris une approche descendante et ont débuté par la définition de l'utilisabilité et ont graduellement raffiné cette définition jusqu'à la caractérisation de propriétés d'utilisabilité (semblables aux spécifications) afin de lier ces dernières à des patrons d'utilisabilité architecturalement sensible (Tableau 2.2).

- *Couche d'attribut*: Le premier niveau de cette composition est ce qui s'appelle attributs d'utilisabilité dans le domaine de IHM. Ces attributs sont des composants précis et mesurables de l'utilisabilité. Les attributs qui sont universellement acceptés sont ceux définis au début de ce chapitre, plus précisément la définition d'ISO 9241.
- *Couche Propriété*: Les propriétés incarnent les heuristiques et les principes de conception que les chercheurs considèrent avoir un impact direct sur l'utilisabilité des systèmes informatiques. Ces propriétés peuvent être utilisées comme requis à l'étape de conception, par exemple en indiquant « le système doit être en mesure de fournir un contrôle des erreurs de l'utilisateur ». Folmer et son équipe ont dérivé leur ensemble de propriétés à partir d'un aperçu étendu du travail de plusieurs auteurs. Cependant, c'est la responsabilité de l'architecte de l'utilisabilité de guider les intervenants dans la détermination d'un ensemble approprié de propriétés.
- *Couche Patrons* : Pour correctement mettre en application les propriétés d'utilisabilité, des patrons architecturalement sensibles sont implantés (Folmer et Bosh 2003). Ces patrons satisfont un certain besoin indiqué par une propriété lors de la définition des besoins du client. Il est important de prendre note que ces patrons ne présentent pas de solution spécifique à incorporer dans l'architecture de l'application, mais ne font que suggérer un mécanisme abstrait qui, une fois conçu, pourrait améliorer l'utilisabilité. C'est alors le devoir de l'expert en utilisabilité d'évaluer si la mise en application d'un tel patron au niveau de l'architecture va permettre l'implantation de la propriété.

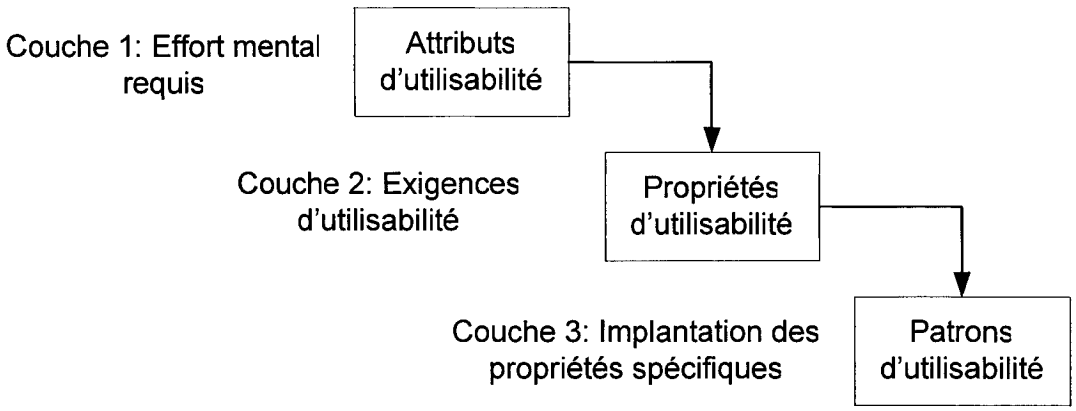
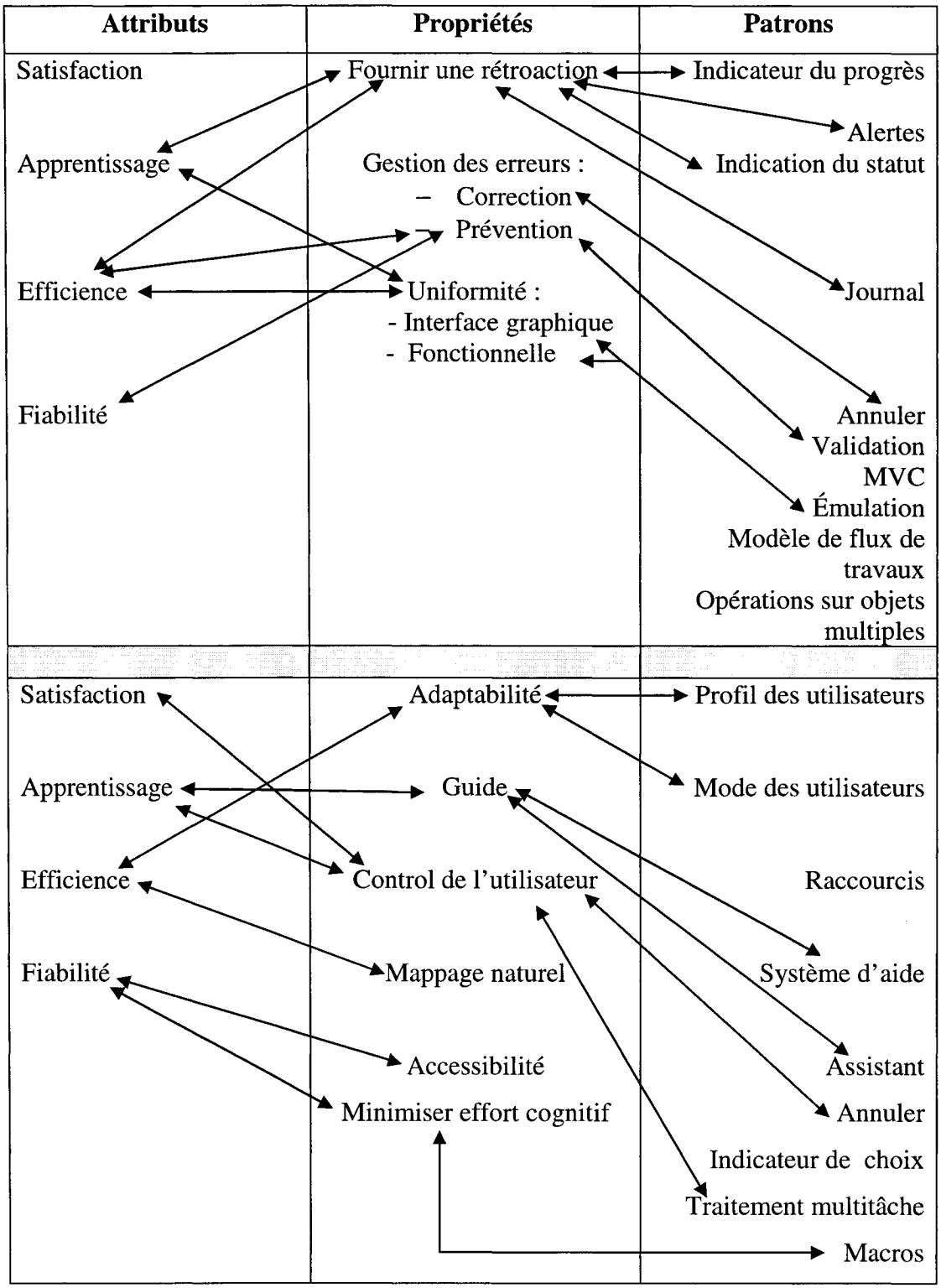


Figure 2.4: Méthodologie de Folmer et al. (2003a)

2.4.2 Efforts à l'institut de Génie Logiciel

Len Bass et ses collègues à l'institut de Génie Logiciel (SEI) (Bass et John 2003) de l'université Carnegie Mellon ont adopté une approche différente en ce qui concerne l'établissement du lien entre l'utilisabilité et l'architecture. Ils ont employé une approche ascendante, en débutant du domaine du génie logiciel, qui permet d'améliorer l'utilisabilité de systèmes informatiques à travers des décisions architecturales. Ils ont identifié des aspects d'utilisabilité avec chaque aspect représenté sous la forme de scénarios qui impliquent plus que la conception détaillée de l'IU et qui a un impact architectural. Par exemple, un scénario couramment employé est lorsqu'un utilisateur effectue une action qu'il n'aimerait plus voir exécutée. Le système devrait alors permettre à l'utilisateur de retourner à l'état initial. Ils ont produit 27 scénarios (Annexe B) et ont proposé par la suite des modèles architecturaux qui suggèrent une solution pour chaque scénario. Il est important de mentionner que les auteurs ne prétendent aucunement que leurs patrons sont les seuls ou même les meilleures solutions.

Tableau 2.2: Projet STATUS (tiré de Folmer, Van Gurp et Bosch 2003a)



2.5 Évaluation de l'architecture dans la perspective de l'utilisabilité

2.5.1 Aperçu

Même si l'architecture du logiciel a récemment émergé comme étant le niveau approprié pour traiter la qualité du logiciel, une attention inadéquate a été prêtée aux méthodes d'évaluation de ces architectures. L'objectif de cette évaluation est d'analyser l'architecture afin de vérifier si certains critères de qualité ont été bien adressés lors de la conception. Plusieurs groupes de recherche ont pris des initiatives et ont proposé de diverses méthodes pour évaluer la qualité de l'architecture (Clements, Kazman et Klein 2001; Dobrica et Niemela 2002). Nous partons de méthodes d'analyse de l'architecture génériques vers des méthodologies plus spécifiques d'évaluation du soutien architectural de l'utilisabilité.

2.5.2 SAAM: Software Architecture Analysis Method

Développée en 1993, la SAAM (Kazman, Abowd, Bass et Webb 1994; Kazman, Abowd, Bass et Clements 1996) est la première méthode à être basée sur les scénarios. Elle est principalement préoccupée par l'évaluation d'une architecture pour la maniabilité (modifiability) tout en indiquant les points où l'architecture échoue à atteindre les exigences de maniabilité. Elle se compose de six étapes principales.

1. *Développement des scénarios*: La première étape est un exercice qui consiste à identifier les types d'activités que le système doit supporter. Ces activités ainsi que les modifications possibles que les intervenants peuvent envisager sont formulées en scénarios.
2. *Description de l'architecture*: L'architecture de l'application est ensuite définie. La notation choisie doit être compréhensible par tous les participants et doit aussi indiquer la représentation statique du système (composants et leurs

interconnections). La description de l'architecture ainsi que son développement s'influencent, il est alors recommandé d'exécuter ces deux étapes en parallèle.

3. *Ordonnancement des scénarios*: Les scénarios sont ensuite classés par ordre d'importance. Ceci est basé sur un vote effectué par les ingénieurs et les intervenants.
4. *Évaluation des scénarios*: L'ingénieur va ici décrire comment l'architecture devrait être changée pour accommoder le scénario. Tous les composants qui sont sujets à des modifications ainsi que l'effort de développement sont identifiés. L'impact architectural du scénario est alors déterminé.
5. *Interaction entre les scénarios*: Lorsque deux scénarios ou plus exigent des changements au(x) même(s) composant(s), on dit qu'ils interagissent l'un avec l'autre. Les composants affectés doivent être alors divisés en sous composants afin d'éviter toute interaction.
6. *Évaluation globale*: Finalement, un poids est assigné à chaque scénario en fonction de son importance au succès du système.

2.5.3 SALUTA: Scenario based Architecture Level UsabiliTy Analysis

Folmer et Bosch (2004) ont récemment effectué un survol des différentes techniques d'évaluation de l'utilisabilité qui ont émergé pendant les dernières décennies. Les plus populaires sont les tests d'utilisabilité (Nielsen 1993), l'évaluation heuristique (Nielsen et Molich 1990) et le *walkthrough cognitif*. Ces techniques peuvent être employées pendant les différentes étapes du cycle de développement.

Malheureusement, aucune de ces méthodes ne met le point sur l'évaluation de l'architecture afin de vérifier le support de l'utilisabilité. En effet, les méthodes d'évaluation de l'architecture traditionnelles, qui ont été présentées dans les sections précédentes, avaient principalement pour objectif la vérification de la maniabilité du système. Folmer et ses collègues (2003b) ont donc pris initiative et ont présenté une première investigation pour l'adaptation d'une de ces méthodes afin de vérifier

l'utilisabilité de l'application. La méthode proposée, SALUTA, se compose de quatre phases :

1. *Création du profil d'utilisation*: Cette étape peut être décomposée en 5 sous étapes.
    a. Identification des utilisateurs cibles.
    b. Identification des tâches.
    c. Identification du contexte d'utilisation
    d. Association des tâches précédentes aux attributs d'utilisabilité définis dans le modèle de Folmer.
    e. Choix et ordonnancement des scénarios
2. *Évaluation de l'architecture*: Deux approches sont définies.
    a. Analyse basée sur les patrons d'utilisabilité: En utilisant la liste de patrons définis dans le modèle, l'objectif est de déterminer la présence de ces patrons dans l'architecture.
    b. Analyse basée sur les propriétés de l'utilisabilité: L'architecture peut être vue comme une série de décisions et solutions de conception (Folmer et Bosh 2002). Les décisions de conception qui mènent à la première version de l'architecture sont analysées afin de vérifier si elles englobent les propriétés d'utilisabilité.
3. *Évaluation des scénarios*: Cette étape consiste à évaluer chacun des scénarios définis dans le profil de l'utilisation et à déterminer les propriétés et/ou les patrons qui permettent leur implantation.
4. *Interprétation des résultats*: Finalement, les résultats doivent être interprétés afin de tirer des conclusions au sujet de l'architecture. Si l'architecture s'avère avoir bien incorporés certains aspects d'utilisabilité, alors le processus de conception est achevé. Autrement, des transformations architecturales doivent être appliquées.

Après avoir exposé des méthodes d'évaluation de l'architecture dans la perspective d'utilisabilité, nous allons maintenant préciser comment les meilleures pratiques d'utilisabilité peuvent être intégrées dans les processus de génie logiciel, et présenter une autre optique de développement centré utilisateur.

CHAPITRE 3

ORGANISATION ET MOTIVATION DE RECHERCHE

Ce chapitre a pour objectif d'introduire les deux articles de revue qui constituent le corps de ce mémoire et d'illustrer la motivation derrière chacune des deux publications et la contribution de recherche qui sera apportée dans le cadre de cette maîtrise.

Le chapitre 4 introduit le premier des deux articles qui forment le corps du présent travail. L'article, intitulé **Investigating the impact of usability on software architecture through scenarios: a case study on Web systems**, par Tamer Rafla, Pierre N. Robillard et Michel Desmarais, va paraître dans la revue *The Journal of Systems and Software* de *Elsevier.*

Cette étude est basée sur le constat que les méthodes traditionnelles d'évaluation de l'utilisabilité précédemment citées (cf. chapitre 2) sont essentiellement axées sur l'évaluation de l'interface utilisateur (IU). Ces techniques s'avèrent insuffisantes pour évaluer l'utilisabilité de systèmes de logiciel, puisque comme certaines recherches l'affirment, des améliorations apportées à l'IU peuvent avoir un impact majeur sur l'architecture. L'utilisabilité d'un système pourrait être considérablement compromise si l'architecture logicielle n'est pas utilisable. Ceci étant dit, des méthodes qui permettent l'analyse de l'architecture pour l'utilisabilité doivent être disponibles aux concepteurs logiciels. Or, une attention inadéquate a été menée vers ces méthodes et les recherches en utilisabilité devraient être orientées vers la définition de ces types de techniques d'évaluation de l'utilisabilité. L'objectif de cet article est de :

1. A ce jour, une seule recherche a été effectuée sur l'analyse de l'architecture pour l'utilisabilité. Ces recherches sont basées sur la méthodologie présentée dans le cadre du projet STATUS (section 2.4.1). De plus, c'est notre conjecture qu'aucune méthode basée sur les travaux

proposés par Bass et John de l'Institut de Génie Logiciel n'a été proposée (section 2.4.2). Une adaptation orientée utilisabilité de SAAM basée sur ces travaux est présentée. Cette méthode va ensuite être utilisée pour déterminer l'impact architectural quand l'utilisabilité est implantée dans un système Web déjà existant. Afin de bien évaluer cet impact, cette analyse sera effectuée sur deux architectures distinctes développées par des étudiants qui ont pris le cours « Atelier de génie logiciel » à l'hiver 2003. Ces deux équipes ont utilisé le *Unified Process for Education* (UPEDU), modèle de processus dérivé du *Rational Unified Process* (RUP).

Le chapitre 5 constitue le second des deux articles. L'article, intitulé **A method to elicit architecturally sensitive usability requirements: its integration into a software development process**, par Tamer Rafla, Pierre N. Robillard et Michel Desmarais, a été soumis pour publication au *Software Quality Journal* de *Springer*.

Comme le premier article de revue l'indique, certaines exigences d'utilisabilité peuvent encourir de sérieux changements sur l'architecture. Ces changements pourraient avoir été réduits ou même évités si ces exigences ont été préalablement définies et considérées, plus précisément avant la phase de conception architecturale. Mais comme peu d'auteurs ont étudié le lien qui existe entre l'utilisabilité et l'architecture, aucune recherche n'a encore été présentée qui permet de guider les intervenants dans l'identification des exigences d'utilisabilité qui sont architecturalement sensibles. L'objectif de cet article est de :

1. Présenter une méthode qui va permettre la définition et l'organisation des requis d'utilisabilité qui peuvent avoir un impact sur l'architecture.
2. Montrer, à l'aide de UPEDU, comment cette méthode peut être modélisée comme une activité au sein de ce processus.

CHAPITRE 4

INVESTIGATING THE IMPACT OF USABILITY ON SOFTWARE ARCHITECTURE THROUGH SCENARIOS: A CASE STUDY ON WEB SYSTEMS

Abstract

Usability has primarily been served by separating the user interface from the remainder of the application. However, several researchers have recently determined that there is a direct relationship between architectural decisions and usability requirements. This leads us to conclude that more attention should be devoted to usability-driven architectural analysis methods. We present a case study, which involves adapting an existing software architecture analysis method (SAAM) for the purpose of deriving the interdependencies between architectural characteristics and usability requirements. More specifically, we investigate the impact on the architecture of implementing usability requirement changes. Potential design solutions that accommodate the corresponding usability mechanisms into the Web software architecture are presented, along with the rationale for applying them and the process by which they are obtained. We conclude by recommending how usability issues can be dealt with proactively during the design of the architecture, and explain the need to integrate those usability requirements into a software engineering process.

4.1 Introduction

Humans play an active and essential role in the operation of interactive software, and the user interface (UI) has become an essential part of many software systems (Nielsen 1993). The architectural answer to building a UI is based on the principles of modularity and separation (Hix and Hartson 1993; Schneiderman 1998; Nielsen 2000).

This separation has made it easier to make modifications to the UI, and usability seemed to be primarily a property of the presentation of information. Indeed, UI had never been considered in software architecture design, due to the widespread assumption among software engineers that it has to do only with the visible part of the system. However, this view is now being challenged (Bass and John 2003). A more recent perspective on the UI is that the software architecture needs to be explicitly designed so that the final system can satisfy usability requirements. Even when the presentation and functionality of a system are well designed, its usability could be greatly compromised if the underlying architecture does not support usability concerns. Moreover, it often proves difficult to make the changes to a system that are necessary to improve its usability. One reason for this is that many of the required usability-driven changes involve changes to the system that cannot be easily accommodated by the software architecture (Folmer and Bosch 2004).

We investigate the link between usability requirements and software architecture in a case study of a Web-based application. The purpose of this paper is twofold: 1) to adapt a scenario-based approach to analyzing a software architecture for usability; 2) to determine how usability requirements can be supported and study their impact on the architecture by proposing Web-specific design solutions.

We begin by discussing the scenario-based method that was used to analyze the Web architecture for usability. Web-specific design solutions are then proposed to determine how usability requirements are supported. Their impact on the architecture is investigated and the results of the late implementation of usability on the architecture are analyzed. We close by discussing how usability issues can be dealt with proactively during the design of the architecture and provide topics of future work.

4.2 SAAM: Software Architecture Analysis Method

The Software Architecture Analysis Method (SAAM) (Kazman et al. 1994; Kazman et al. 1996; Clements 2001) was among the first to address the assessment of software

architectures using scenarios. SAAM has traditionally been applied to the development of related software qualities such as maintainability (Haggander and Bengtsson 1999; Bengtsson and Bosch 2000; Dobrica and Niemela 2002). Since very few architectural assessment techniques for usability exist, can SAAM also be considered for usability? We explore this avenue after briefly describing the method. SAAM, (Figure 4.1) consists of the following four steps:

- *Scenario development*: The first step is to illustrate the kinds of activities the system must support and the kinds of anticipated changes that will be made to it. Since usability is often defined in a very abstract fashion and is not well interpreted by software designers, scenarios are a good way of synthesizing individual interpretations of software quality into a common view, which can make usability requirements more specific.
- *Architecture description*: If there is no description of the system architecture, then the product is thoroughly studied and an architectural diagram is developed. This activity is to be carried out in parallel with the previous activity in an iterative mode. The final version of the architecture description, together with the scenarios, serve as the input for the subsequent activities of the method.
- *Scenario evaluation*: SAAM evaluates a scenario by investigating which architectural elements are affected by that scenario. The impact of the modifications associated with each scenario is estimated by listing the components that are affected.
- *Overall evaluation*: Finally, following scenario evaluation, the results are interpreted to draw conclusions concerning the software architecture and to assess the impact of the usability scenarios on the architecture.

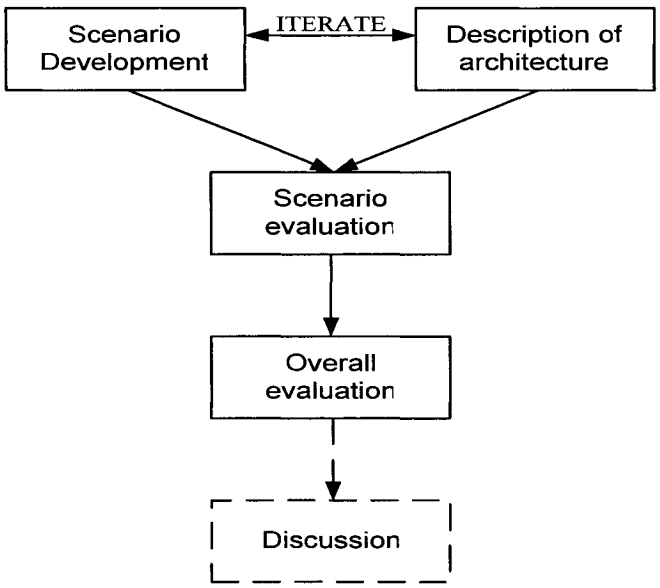


Figure 4.1: Steps of the SAAM (adapted from Kazman *et al.* 1994)

In the sections below, the usability-driven analysis of the architecture is conducted by elucidating the four SAAM phases.

4.3 Scenario development

In this section, step 1 of the method is performed. Bass and his team at the Software Engineering Institute identified a collection of architecturally significant scenarios, which expressed a usability issue (Bass et al. 2001). Those scenarios are common to many interactive systems and are not related to the domain functionality of any system. The ones that will inevitably increase the usability of the system serving as the case study for this work are extracted, and their respective usability benefits are illustrated (Table 4.1). These benefits encompass the following: indicators of efficiency, problem-solving and learnability (effectiveness) and customer satisfaction (Constantine and Lockwood 1999; Shackel 1991; ISO 9241 1998). These scenarios are applicable to Web applications and can be considered for other Web systems suffering from the same

usability issues identified here. However, we do not claim that this set of usability scenarios covers all the usability problems identified in Internet applications.

- *Checking for correctness*: Users may make errors they do not notice. Depending on the context, error correction should be enforced directly or suggested through system prompts. Mistakes made entering credit card billing information, for example, can require costly manual follow-up. The checking-for-correctness scenario will be validated if and only if a mechanism allowing for all entered fields to be validated is incorporated.
- *Retrieving forgotten passwords*: Users may forget their passwords, and retrieving them may be difficult or may cause lapses in security. Systems should provide alternative strategies to grant the user access. It becomes tedious to contact the site administrator every time a password is forgotten. Since this is not always accomplished in a timely fashion, a user should be able to retrieve it without help. Thus, retrieving forgotten passwords is another scenario worth considering.
- *Modifying interfaces*: Poorly designed interfaces increase user errors. Looking at the system from a user's point of view reveals that several important aesthetic issues that need to be considered to make the site more usable, are not addressed. Good hypermedia design practices should be used to develop applications that are easy to use, provide friendly navigational spaces and seamlessly integrate the underlying transactional behavior.
- *Providing good help*: It does not matter how intuitive an application is; if it is rich in functionality, then the user will inevitably need some kind of assistance. Moreover, as Web applications become more complex, the importance of providing help grows inexorably. Since the system is not equipped with a help system, and considering that some fields are difficult to comprehend, providing good help is another scenario that should be taken into account.
- *Supporting international use*: The increasing diversity of the Internet has created a tremendous number of multilingual resources on the Web. Users may want to

configure an application to communicate in their own language or according to their cultural norms. Systems should be easily configurable for deployment in multiple languages.

4.4 Architecture description

4.4.1 Replan: The meeting management system

Implemented using the Unified Process for EDUcation (UPEDU) (Robillard et al. 2003), Replan is a Web-based meeting management system aimed at the organizers of meetings where the numbers of participants and their geographic distribution make scheduling difficult. This system would allow meeting coordinators to send availability requests to a set of individuals, so that each of them can specify personal availability periods. The set of availability periods would then be graphically represented using a calendar tool to enable a coordinator to visualize the relevant information at a glance, making the scheduling decisions easier.

4.4.2 Replan's architecture

The Java Enterprise Edition (J2EE™) Web architecture (Inderjeet et al. 2002) consists of Web browsers, a Web server and a network connection (Figures 4.2 and 4.3). Browsers request Web pages from the Web server, which distributes JavaServer Pages (JSP) of formatted information to clients. This request is made over a network connection and uses the HTTP protocol.

Table 4.1: Linking usability aspects to scenarios (Bass and John 2003)

Usability Aspects / Scenarios	Effectiveness				Efficiency		Satisfaction
	Accelerates error-free portion	**Reduces impact of slips**	**Prevents mistakes**	**Accom-modates mistakes**	**Supports problem-solving**	**Facilitates learning**	**Increases confidence and comfort**
Providing good help					√	√	
Modifying interfaces	√			√			
Checking for correctness		√	√	√	√		
Retrieving forgotten passwords				√			
Supporting International use	√	√	√	√	√	√	√

In this implementation of the architecture, the JSP files not only contain the HTML code necessary to produce the UI, but are also responsible for processing the user's requests in order to perform the necessary business logic. Hence, the JSP contains a significant amount of embedded Java code responsible for storing the session and linking JSP pages together. The JSP component is in charge of request processing and the creation of any objects used by the JSP, as well as deciding, depending on the user's actions, to which JSP page the request is to be forwarded. Functionalities of this component include the validation of request parameters with JavaScript, the authentication of the user and the posting of user forms by connecting to the database through the Java component, before dispatching the request to another JSP for presentation of the response. The Java component of the system interacts with an Oracle database. The Tomcat Web server combines the components according to configuration information specifying the connection between these components. In fact, interdependent JSPs have been grouped into modules since the system is modeled from a high-level point of view and the relationship between those JSPs is not of interest to us. The application control is decentralized, because the current JSP page being displayed determines the next page to display. The page flows are embedded on the links between the pages.

We wish to verify whether or not the incorporation of usability requirements would be possible with different implementations of the Web architecture. Two architectures were selected for the following reasons:

- To ensure the validity and correctness of the results.
- To show that the proposed solutions are possible with different implementations of the architecture.
- The other excluded projects suffered major drawbacks, such as failure to meet the requirements. Some features were nonexistent and the system was not fully operational.

Table 2 compares the two selected architectures, comparing the number of Java and JSP files since they are the essence of the J2EE Web architecture. The number of tables

created in the database is also illustrated. Team A has 29 Java classes, accounting for 160 KB, and 15 JSP files, accounting for 50 KB. The business logic represents 76% of the overall structure of the system, while the interface represents only 24%. This first system was properly implemented, since business logic is well encapsulated in the Java classes and the JSP files were used only to embed some HTML code for the UI. However, Team B has more JSP files than Java classes. It has 24 JSP files for a total size of 200 KB, but only 7 Java classes for a total size of 74 KB. The business logic represents only 26% of the overall structure of the system, while the view represents 74%. Some business logic is incorporated in the JSP component rather than in the Java component, and the guidelines of for Web architecture design were not properly followed. Figures 4.2 and 4.3 illustrate the architecture of the two selected systems.

Table 4.2: Structure of two versions of Replan

	Java		**JSP**		**Database**
	Number	**Size (KB)**	**Number**	**Size (KB)**	
Team A	29	160	15	50	5
Team B	7	74	24	200	11

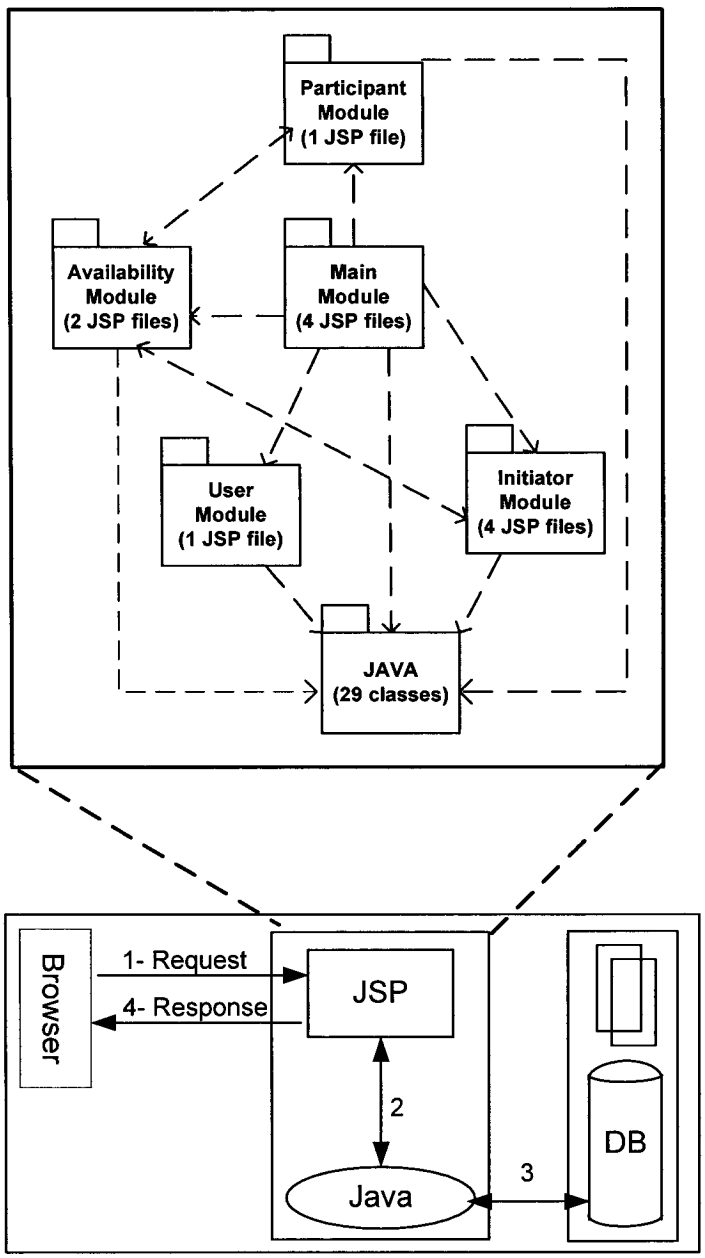


Figure 4.2: Team A's architecture

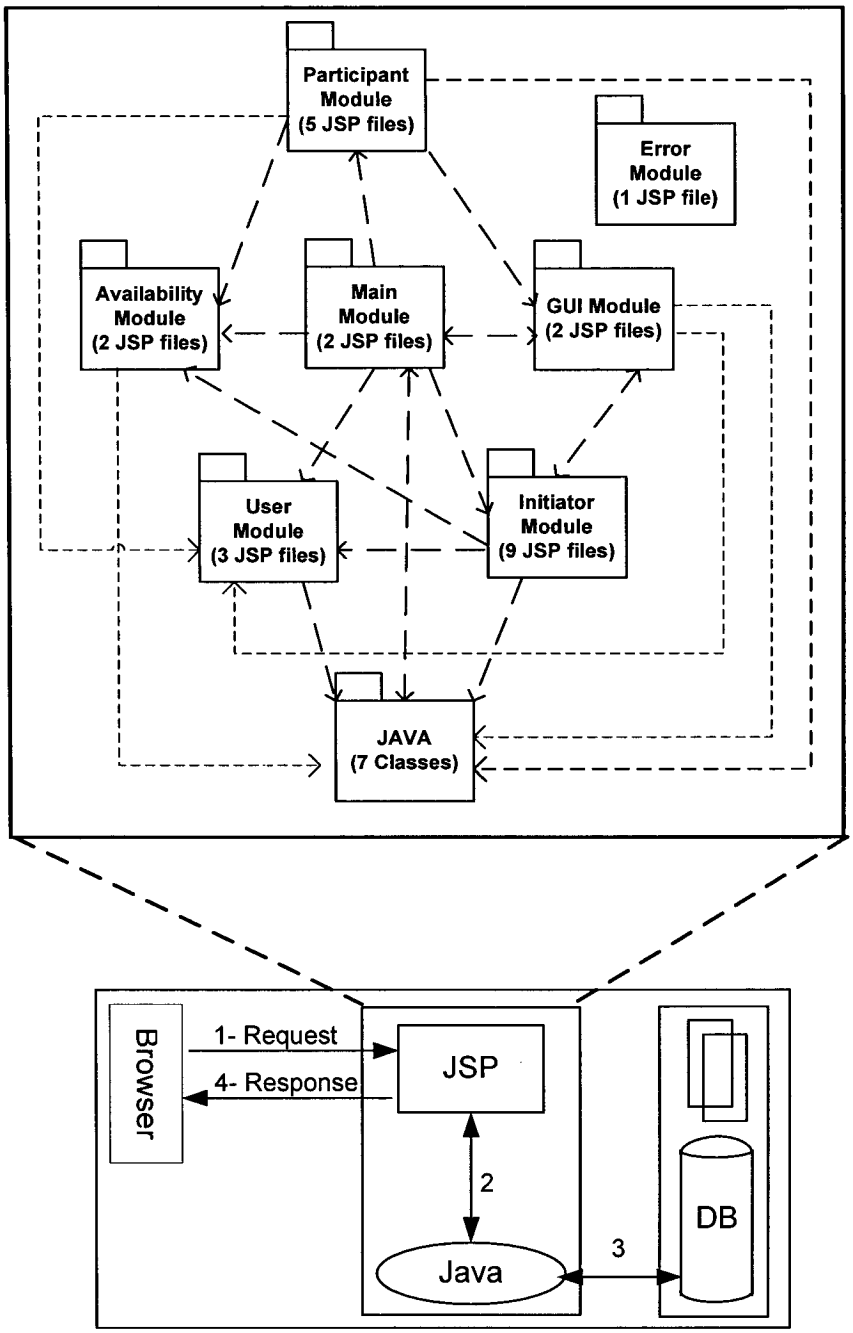


Figure 4.3: Team B's architecture

4.5 Scenario evaluation

In this section, step 3 is exemplified. Design solutions to satisfy the chosen usability scenarios are presented. The magnitude of the impact is represented as follows:

- *No impact*: A small number of Web components are modified for the two selected architectures and the development effort is low. The required changes are categorized as architecturally independent and can still be implemented once the system has been designed.
- *Minor impact*: The number of modified architectural components is different; however, the scenarios necessitate more or less the same implementation effort. Nevertheless, it is beneficial to consider those usability requirements at earlier stages of development.
- *Major impact*: Sizeable components are reworked, and the required changes necessitate a substantial development effort. This effort is entirely dependent on the implementation of the architecture. Such modifications should be considered at the design stage due to the considerable effort they entail.

4.5.1 Checking for correctness

With server-side validation, the processing of the form fields is performed after they are submitted to the server. Using JavaBeans (Inderjeet et al. 2002), the designer needs to create a "validation bean" that will perform secure server-side validation on the entered data. Implementing a validation bean requires several steps. First, a data bean that has fields identical to the form fields must be designed. Second, the validation rules for these fields must be devised. These two steps are implemented in the Java component of the application. For example, if the validation of the user's form is required, then a validation bean with its respective validation rules needs to be added to the User Java class. Finally, all JSP files must be redesigned to work effectively with every validation bean that was introduced. They are slightly modified to incorporate the

few lines of code responsible for initializing the bean and directing the request to the Java classes for validation. This approach separates the business logic (Java component) from the presentation (JSP component).

This approach of providing form validation entails the modification of the same number of components and files for both architectures (Figures 4.4 and 4.5). Surprisingly, after detailed analysis of the two architectures, it became evident that the JSP contains some embedded Java code, which is responsible for storing all objects and also for transferring control to another JSP in charge of validation. Unfortunately, business logic should be encapsulated in the Java component, while the presentation resides in the JSP component. This means that all the inserted Java code in the JSP file should be moved to the Java component and the JSP file should contain only the HTML code responsible for displaying the form. However, this method involves a significant development effort in both architectures, since a large number of components are reworked. The impact of this scenario on the Web architecture is classified as intermediate.

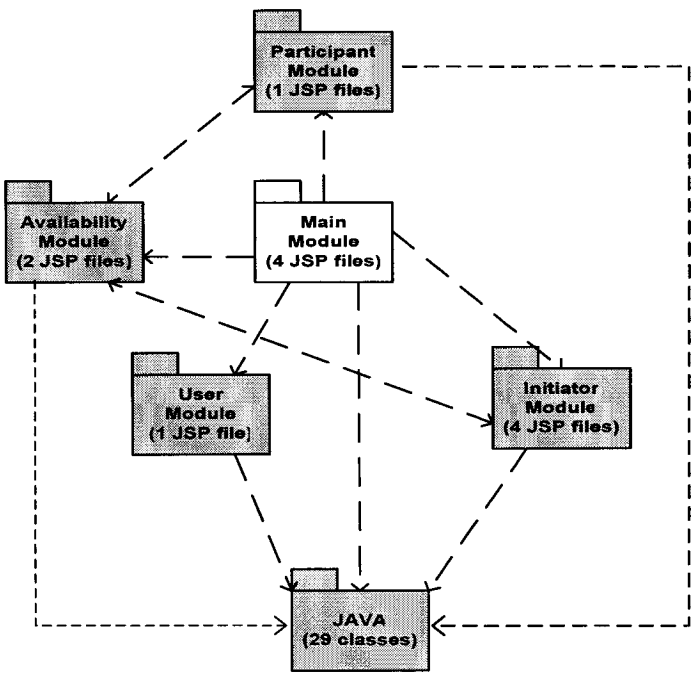


Figure 4.4: Team A – server side validation

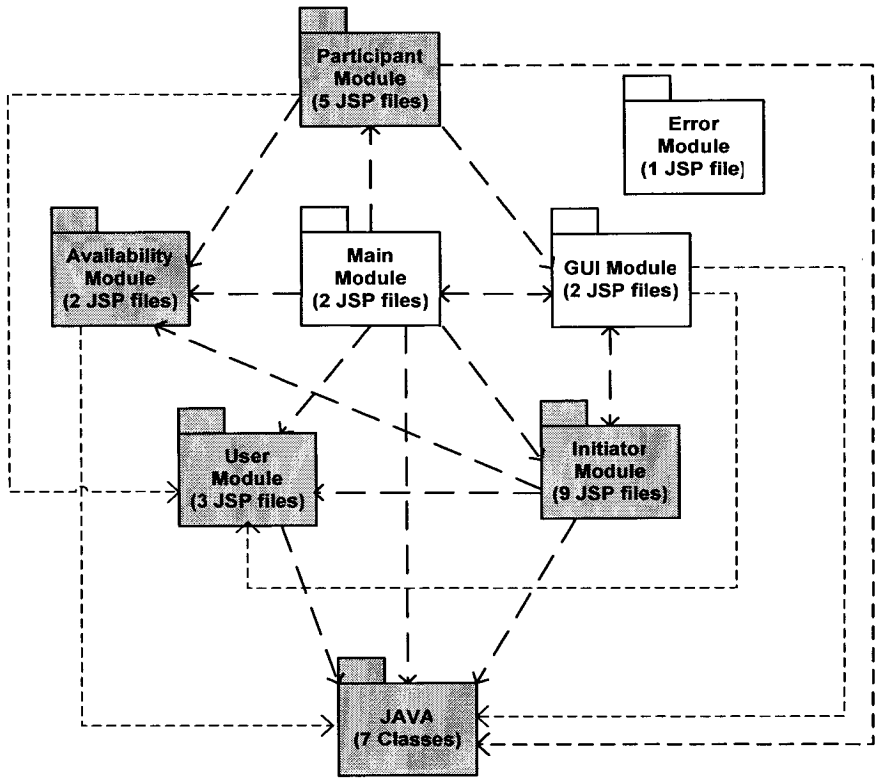


Figure 4.5: Team B – server side validation

4.5.2 Retrieving forgotten passwords

The Web has moved beyond purely open content available to all. It is now commonly used to provide information that ought to be restricted in some way to members. One common method of restricting access is to ask users to enter a name ("username") and a password. This simple combination can be a source of annoyance and frustration to users because the user's password can be forgotten. Adding a link in the "login" page that will redirect the user to a "lost password" page where the user can enter his/her username is the simplest solution. Once the request is submitted, the password is emailed to the user. This procedure is fairly simple to implement since the username is basically the email of the user according to the specifications of the system, and the modifications are reasonably straightforward to implement (Figure 4.6). A few

lines of Java code were added to the User class (Java component) to select the user's password from the database. Also, a *LostPassword* JSP page, which is linked to the *Login* JSP page, is created to email that password to the user. That newly created JSP is included in the Main module with the *Login* JSP file. The changes required to instigate this scenario require the developer to rework the JSP and the Java components (Figure 4.6). Fortunately, these adjustments seem to be rather simple to make, since they do not require a significant programming effort and can be added at any stage of development. This solution works for both architectures. The Java component has been slightly modified for both teams and a new interface has been added to the Main module in the JSP component. The J2EE architecture can easily accommodate the mechanism for retrieving forgotten passwords. This scenario does not have an impact on the system's architecture, since it requires a low development effort for both teams and is independent of the overall implementation of the system.

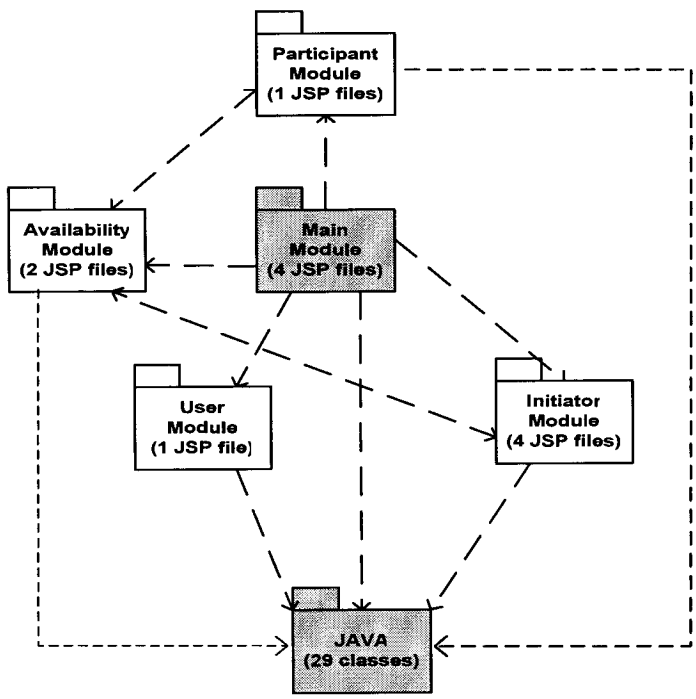


Figure 4.6: Team A – retrieving forgotten passwords

4.5.3 Supporting international use

In today's competitive world, many businesses are becoming global in order to reach the widest possible targeted audience. This is a problem, because Web applications developed by these organizations must be suitable for customers who use and understand different languages. Therefore, there is an increasing need to develop and design cost-effective Web solutions capable of delivering content in multiple languages. To be considered "multilingual", an application has to support the following elemental categories of features:

- *Data representation*: supporting all the character set encodings that an end-user may use.
- *Data display*: displaying strings, currencies, numbers, date formats, weights and measures, etc. in a format an end-user will understand.
- *Data input*: the system must understand the format of the data being submitted to the application, this knowledge is essential for data integrity; for example, a French-speaking user will submit the date "9/11/04" for November 9th, 2004 (dd/mm/year), while an American English-speaking user will submit that date as 11/09/04 (mm/dd/year).

The internationalization of data-intensive Web applications can be achieved by replicating the system. This means that it is necessary to replicate the database, translate the HTML code embedded in the JSP files, and modify the Java code to manipulate the newly introduced set of encodings. The required translation language is assumed to be alphabet-based, and languages, which use symbols are not, considered (e.g. Chinese, Arabic, etc.)

4.5.3.1 Database extension and query rewriting

The main characteristic of a data-intensive Web application is the interaction with database servers, which dynamically provide the data to be published in the requested pages. The internationalization process requires modification of the information stored

in the database, thus an extension of the database schema is necessary in order to support such translation.

- For each table *T*, representing an entity type with attributes subject to translation, a new table T_{trans} has to be created with T_{trans} = { PK_T, language, { A_{t_1},...., A_{t_n} } }
- For each table *R*, representing a relationship type with attributes subject to translation, a new table R_{trans} (PK_R, language, A_{t_1},...., A_{t_n}) has to be created.
- (PK_T is the set of attributes representing the primary key, and { A_{t_1},...., A_{t_n} } is the set of new attributes)

Rewriting the queries is indispensable in order to include the modified attributes in the result of each query. The characteristics of this rewriting process are as follows:

- The system will be used by English speaking users. This language is obtained by applying a custom rule to the preference language list coming from the HTTP request. French is denoted as the master language and English as the requested language.
- Each attribute *A* subject to translation is replaced with two attributes French-A and English-A containing its value in French and English respectively.

Extending the database and rewriting the queries would solve the data display, the data representation and the data input issues. Only an additional table for each entity or a relationship-type object to be translated is added. In the worst-case situation, according to Aykin (1999), given a database containing *m* tables with attributes subject to translation, the number of new tables that have to be added is equal to *2m*. Thus, all the database tables should be duplicated, and the queries nested in the Java component should be reworked for both systems (Figure 4.7).

4.5.3.2 Translating the web pages

The object of rewriting the queries is to extract from the database the requested data together with their translations in the desired language. All JSP files are thoroughly

examined and all keywords are translated. Considering that English is the requested language, all attributes should be available in that language. The required modification does have a major impact on the architecture of Replan. A different implementation effort would be necessary to incorporate the *supporting international use* scenario in both teams, since their JSP, Java and database components are different and the development effort is reasonably high (Table 4.2).

4.5.4 Modifying the interface

Usability is no longer a luxury in Web development. As the popularity and acceptance of Web-based systems increase, the issues related to usability and UI design are increasing in frequency. While people may grow comfortable and reassured with respect to one of the most obvious issues of concern, security, they may find it difficult to feel reassured about some Web applications because of their lack of standards and practices associated with usability. By applying Nielsen's usability heuristics (1994; 2002), the daunting task of improving features in a data-intensive Web application can be addressed. Using this set of heuristics as a touchstone in the evaluation process, it becomes clear that the system suffers from usability issues such as the absence of a title or a header describing screen content, and an inconsistent icon design scheme and stylistic treatment across the system. The design of the JSP files that contain the HTML fragments making up the UI should minimize the user's memory load by keeping the same navigation bar and page layout. A flexible, minimalist aesthetic design is easier on the users' eyes and allows them to concentrate on the task at hand without distraction in the interface design (odd colors, for example). The use of a linear progression facilitates a narrative flow, as well as giving the user a feeling of control over the process. Matching the system and real-world expectations means that, when the task is "completing a form", the flow in completing that form should be a process as smooth and problem-free as possible.

Since this scenario is solely concerned with the presentation of information, separating the UI from the remainder of the application is sufficient to resolve all the violated heuristics. More specifically, this scenario can be satisfied by looking at the JSP files that suffer from the usability issues revealed through the application of Nielsen's usability heuristics, and then reworking those files. That being said, the modifications that will be brought to the UI can be implemented on an existing system at a late stage of development. Even though this scenario requires the modification of only one component and has no impact on the architecture, we decided to classify it as minor. UI design expertise is often nonexistent during development, and the time-consuming aesthetic modifications that the development team implement in the end could have been reduced, or even avoided, if the user had been involved in its design at the outset (Seffah et al. 2005). The objective here was not to list all the violated heuristics, but merely to state that they can be corrected without affecting the architecture of either application too severely.

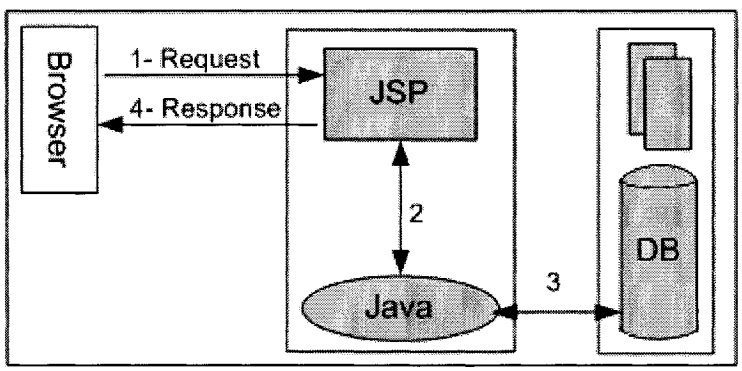


Figure 4.7: Team B – international use

4.5.5 Providing good help

Users do not tend to read printed documents before using a Web application, so online help needs to be easy to use and allow quick location of the necessary information. A "help system" should be well organized and should present users with a clear path to the information they need, either through context-sensitive help or by

providing intuitive and advanced search capabilities. Since most Web applications are simply several pages linked together, if the user has a question or requires help, a simple hyperlink to the FAQ section usually does the trick.

There are many different types of web application help systems. By far, the most popular method is simply to create one page per help item for every page on the website. Basically, the information required is all on one page. Users click on the hyperlink and obtain access to everything they could ever want to know about the site. A problem arises when they simply have a quick question to which the help option should generate a quick and simple answer, and yet they enter a world where they must scroll through massive amounts of data in the hope of finding the answer to the now forgotten question.

The other, more advanced method is a full-blown help application. Once the help icon is clicked, the user dives into an application for the help system. This type of application is more in common with online books. It is more thorough because it covers the entire application, including other aspects such as customization or the installation of client components.

A solution, which lies between the two in functionality, is proposed. This is an easy design solution where each page on the site has its own separate help page accessed by calling up the *Help* JSP file, and which is going to be added to the JSP component. Every Web page would include this simple hyperlink (Figure 4.8). The newly introduced page will, once accessed, extract the page name that called the help system and store it in a variable. That variable will then look up the corresponding help file in HTML format in a newly created table in the database that contains an index on the calling page's name field. Once retrieved, the help text will be inserted into the JSP file and will be presented to the browser. A "back" button, containing the link to the referring page should also be added, to return the browser to the original page that initiated the entire help journey. The database will contain one row in a table for every page. Each row has all the required text to be displayed for the application. Also, if the help text changes, all that is required is an update to the table row for the specific page's

entry. There are no pages to change, and it is a clean and simple way to integrate a help function into the system. This solution is adequate because the Tomcat Web server can support server variables (such as REFERER_URL), which will notify the target page of the name of the originating JSP document. The implementation of this scenario is dependent on the size of the JSP component, more precisely the number of JSP files. According to Table 2, Team A has 15 JSPs, while Team B has 24. The greater the number of interfaces (JSP files), the bigger the help system, since every interface should have its own help content. The addition of a help system has a minor impact on the architecture of Replan. The effort needed to incorporate this scenario is entirely dependent on the actual implementation of the JSP component and the database. This entails doing nothing to the Web application, since the help system is completely self-contained.

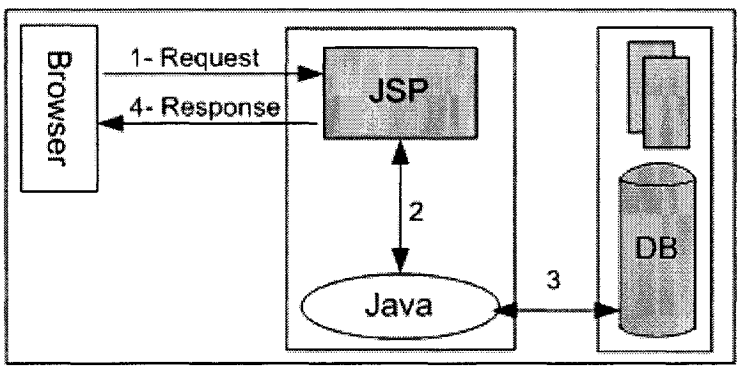


Figure 4.8: Team B – help system

4.6 Overall evaluation

4.6.1 Summary

If the required development effort is low and is the same for both Team A and Team B, then we conclude that the required modifications are architecturally insensitive, since the two teams have distinct architectures. The *retrieving forgotten passwords* scenario seems to be independent of the architecture, and the proposed solution would

necessitate the same effort for both systems. The Java component will be modified to retrieve the user's password from the database, and a lost password page (added to the Main module) is created. Fortunately, these modifications will leave the architecture unchanged, minimizing the impact on the architecture.

If the envisioned usability modifications require the reworking of a relatively small number of architectural components, then the changes will have a minor impact on the architecture. In this case, the required changes necessitate a fairly small development effort. The *modifying interfaces* scenario necessitates the listing of all the interfaces that are lacking important usability guidelines established by Nielsen, and correcting them. In fact, if UI design expertise was present during architectural design, Nielsen's guidelines would have been properly followed and the effort required to go through all the interfaces could have been avoided. Also, for the *providing good help* scenario, a new table containing the help information has to be created. This requires only the modification of the database, and not the modification of major components. All JSP pages should be redesigned to include a link to the new Help page. The greater the number of JSP files, the larger the new Help database table. Both of these scenarios entail a minor impact on the architecture, given that it requires roughly the same development effort for both teams and does not involve the revision of major components.

In order to implement the *checking for correctness* scenario, a validation bean responsible for performing secure server-side validation on the data entered in the form is added to the Java component. For example, if the validation of the user's personal information form is required, then a validation bean with its respective validation rules is added to the User class. According to the J2EE development guidelines, business logic should be encapsulated in the Java component while the presentation would reside in the JSP component. In this implementation of the Web architecture, the JSP component contains some embedded Java code. To implement the validation bean, all inserted Java code in the JSP should first be moved to the Java. The development effort

is rather significant and the architectural impact of the *checking for correctness* scenario is classified as intermediate.

The *supporting international use* scenario primarily involves modification of the database. Every table in the database is subject to replication, since data input and data display in an appropriate language should be ensured. The greater the number of tables, the greater the effort to validate that scenario. Team A has five tables, while Team B has 11. Thus, the effort required to develop the database is considerably greater for the second team. Unfortunately, extending the database does not mean that the scenario is satisfied, as the JSP and Java components need to be revised as well. All interfaces should be translated. Generating a large number of JSP files would require a significant effort to translate the HTML fragments embedded in those files. Also, the database queries need to be rewritten to manipulate the newly introduced parameters in the database. In conclusion, the *supporting international use* scenario has a major impact on the architecture of Replan, since its implementation requires a substantial effort in terms of integrating it into two noticeably different architectures, and all the components of the architecture are modified. Table 4.3 lists the chosen scenarios and their impact on the Web architecture.

Table 4.3: Results of scenario evaluation

Usability Scenarios	Architectural impact	Files affected	Components modified
Checking for correctness	Intermediate	• 4 JSPs that contain forms • 3 Java classes	JSP and Java
Retrieving forgotten passwords	No	• 1 JSP file (Help.jsp) • 1 java class (User.java)	JSP and Java
Providing good help	Minor	• All JSP files • Database tables	JSP and Database
Modifying interfaces	Minor	• All JSP files suffering from usability issues	JSP
Supporting international use	Major	• All JSP files • All Java classes containing SQL queries • Database tables	JSP, Java and Database

4.6.2 Discussion

The *retrieving forgotten password* scenario, which covered only one indicator of the effectiveness aspect of usability (accommodating mistakes) (Table 1), has no impact on the architecture. The *providing good help* scenario, which covers both indicators of the efficiency aspect of usability (supporting problem-solving and learnability), and the *modifying interfaces* scenario, which covers accelerating error-free portions and accommodating the mistake aspects of effectiveness, necessitate minor modifications to the architecture. The *checking for correctness* scenario covers four usability indicators, encompassing both the effectiveness and the efficiency aspects, has an intermediate impact on the architecture. The *internationalization* scenario supports all three aspects (effectiveness, efficiency and satisfaction) and has a major impact on the architecture. In addition, this scenario is the only one that supports increasing the user's confidence and comfort. Hence, as shown in Figure 4.9, there is a relationship between the number of usability aspects of a scenario and the impact it will have on the architecture. The greater the number of usability aspects, the greater the impact of the scenario on the

architecture. This work can be useful to a design team in evaluating and designing a Web architecture. It would enable them to select and implement the scenarios that can easily be incorporated into the architecture (the ones that cover a low number of usability aspects) at later stages of development. It would also allow them to consider, during the elaboration of the architecture, scenarios that require the overall structure of the system to be reworked. Those scenarios are said to be "complex", since they cover many cognitive aspects, such as increasing individual effectiveness, reducing the impact of mistakes and increasing the comfort level of the user.

Also, most of the chosen usability scenarios can be integrated into both existing systems without incurring major changes to the architectures, except for one, the *supporting international use* scenario. This is the only scenario that requires all Web components, as well as the database, to be reworked. It does not entail the same implementation effort for both systems. Having a greater number of database tables means that the effort for updating and maintenance is higher, since every instance in that database has to be maintained. Therefore, the *supporting international use* has an impact on another quality attribute, maintainability. Regrettably, this last scenario is not the only one that has an impact on maintainability. If the separation of concerns is not established, as it should be, then the *checking for correctness* scenario also has an impact on that quality attribute. Problems can arise when applications contain a mixture of data access code, business logic and presentation code. Such applications are difficult to maintain because the interdependencies between all the components can cause strong ripple effects whenever a change is made at any given location. The Web components are difficult to reuse, as they depend on many other classes and require maintenance in multiple places.

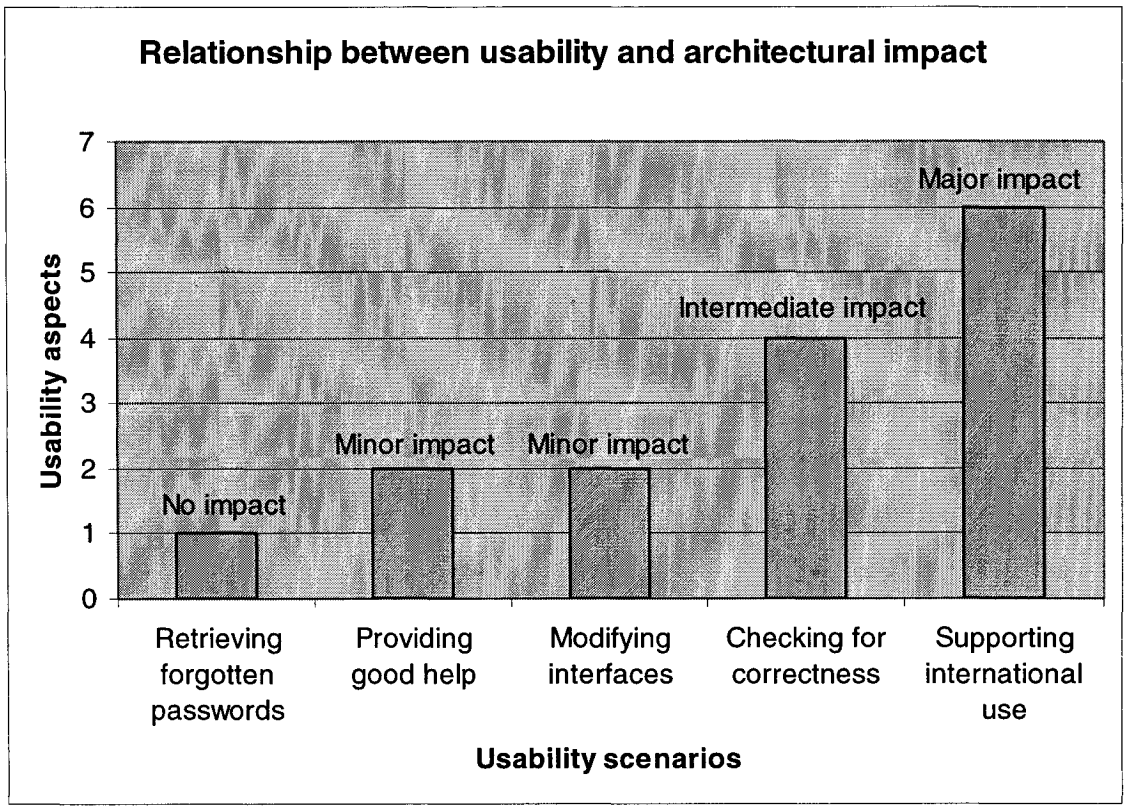


Figure 4.9: Relationship between usability and architectural impact

4.6.3 Recommendations

Recent studies confirm that a significant part of the maintenance costs of software systems is spent on usability issues (Li and Henry 1993). These high costs are largely due to the usability requirements, which will not be revealed until the software has been implemented or deployed. This analysis shows that the usability scenarios that entail a substantial architectural impact (checking for correctness and supporting international use) have an effect on the maintainability of a Web application. The system architecture was not designed to cope with the incorporation of late usability changes. We firmly believe that the architectural impact of those chosen usability scenarios could have been less serious if the architecture had been different. We discuss, in the following section, how the scenarios that entail a significant impact on the architecture, namely *checking*

for correctness and *supporting international use*, can be instigated and their architectural impact minimized.

In any Web-based applications, the data can be validated in two ways. The first method is the approach that was discussed previously, validation on the server side, while the second is to validate the data on the client side, before any of it is submitted to the server. This is usually achieved with JavaScript (JS) running on the client's browser. A few JS functions are created which handle common validation tasks. Activated by clicking the form's "submit" button, they test field content for proper formatting. If they detect a problem, the submission is aborted and an error message is displayed. Client-side validation has numerous advantages for both server and user. It lightens the load on the server and allows the data to be validated before it is sent for processing. Since validation takes place on the client's machine, there is no delay incurred by the client contacting a remote server. Besides the user getting a quicker response, server-processing power is conserved. The implementation of this approach in the architecture would require transferring the control responsible for validation to an external component responsible for regrouping all JavaScript validation files. Figure 4.10 illustrates the proposed new architecture.

The main component that needs to be reworked in order to satisfy the *supporting international use* scenario is the database, as it is closely coupled to the database queries in the Java component. If this scenario was introduced during the architectural design phase, would the design of the database have been different? The software architecture is the artifact that embodies the earliest design decisions, and, if this scenario had been introduced during the architectural description phase, then we definitely believe that the system would have been designed in a way that would accommodate that scenario. Both teams would eventually have spent more time on the design of their database, since this has an effect on their development effort of the Java component. This scenario should have been related to decisions taken early in software development process. Hence, since architectural decisions can preclude the delivery of a usable system, we strongly believe that the inclusion of architecturally sensitive usability scenarios in an

architecture-centric software development process such as UPEDU could help us design more usable systems

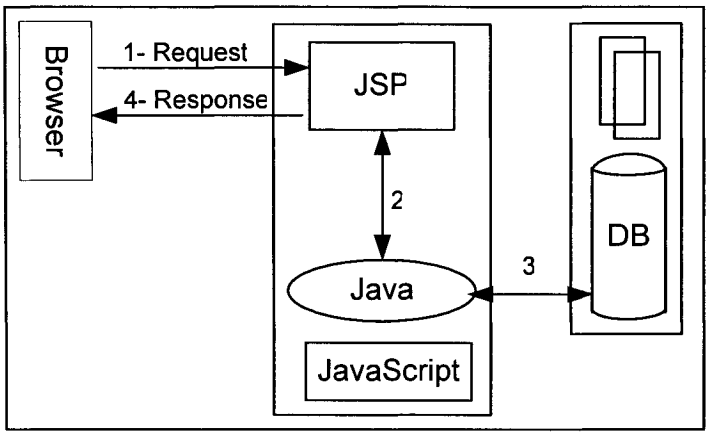


Figure 4.10: New proposed architecture

4.7 Conclusion and future work

In the software and usability engineering communities, little work has been presented on how to integrate usability in the design of software architectures. We contribute to this ongoing research by discussing how SAAM, a scenario-based architecture analysis technique, can be adapted to assist software designers in designing software architectures, which support usability. Our application of SAAM consists of four major steps: (1) determining the set of usability scenarios, (2) describing the software architecture, (3) scenario evaluation, and (4) overall evaluation and discussion.

The first step of the method consisted in extracting from the work of Bass et al. the scenarios that are applicable to Web applications, along with their respective usability benefits. Potential Web-based design solutions were then proposed to highlight the impact of those scenarios on the Web architecture, once instigated. The proposed solutions were obtained by means of an inductive process, which guarantees that these solutions, albeit not necessarily the only ones, are possible. After analyzing two distinct architectures of the same system, our study shows that four of the six chosen scenarios

could be implemented in an existing system without incurring major architectural changes. The usability scenario that covered one usability benefit (namely retrieving forgotten passwords) has no impact on the architecture, while the ones that covered two benefits (such as the modifying interfaces and providing good help) have a minor impact. In fact, the scenarios that revealed a large number of usability aspects (checking for correctness and supporting international use) have a major impact on the architecture. There is, therefore, a relationship between usability scenarios and their architectural impact. Moreover, besides having a serious impact on the architecture, these previous scenarios seem to have an effect on the maintainability of the system if they are implemented at later stages of development. This effect could be avoided if the usability requirements are defined and considered in the architectural description phase. Both teams constructed their architectures without paying sufficient attention to usability requirements and whether or not they can easily accommodate late modifications. This is why relating system usability to architectural decisions taken during the design phase is crucial, and the architecture should be designed to incorporate usability requirements. Software development processes, which have evolved from human-centered design or human-computer interaction, like ISO standard 13407, do not define software architectures, and therefore the usability scenarios that have an impact on the architecture cannot really be implemented. By contrast, since UPEDU is defined as architecture-centric, the next step of this research will be to explore how it can accommodate user concerns and also to investigate how the architecturally sensitive usability aspects can be integrated into the process.

CHAPITRE 5

A METHOD TO ELICIT ARCHITECTURALLY SENSITIVE USABILITY REQUIREMENTS: ITS INTEGRATION INTO A SOFTWARE DEVELOPMENT PROCESS

Abstract

In the software engineering community, it is widely accepted that usability is one of the key quality attributes to be considered in software development. It has primarily been concerned with the presentation of information, more precisely with the user interface. However, some usability problems can prove costly to fix if the changes require modifications that reach beyond the presentation layer, namely those that cannot be easily accommodated by the software architecture. Taking into account some usability requirements earlier in the software development cycle, more specifically before architectural design, can reduce the cost of these modifications. There is scarcity of methods and guidelines with the scope of directing users in eliciting the usability requirements that can impact a software architecture. In this paper, we propose a usability driven adaptation of the quality attribute workshop (QAW) to assist software development organizations in discovering and documenting those usability requirements. We then show how this method can be integrated into an existing software development process. An exercise conducted to assess the utility of this method, reveals that participants were successful in identifying the architecturally relevant usability requirements. It also helped discern the usability aspects that would not have been necessarily defined if this technique had not been employed.

5.1 Introduction

As the market is becoming saturated with competing software products, an increasing number of companies are focusing on delivering usable software systems. This has lead to usability becoming more popular and widely recognised as important quality attribute (Bevan, 1995; Landauer, 1995; Juristo et al., 2001; Abran et al., 2003).

Up to this point, usability has essentially been perceived as a property of the presentation of information and not been a concern for software designers beyond separating the user interface from the remainder of the application (Nielsen, 1993). That approach is commonly used in practice but it has shortcomings. These problems are, first that there are some usability requirements, which refer to the specification of the required usability for the software that are not taken into account by separation as they might impact the architecture of the system (Bass and John, 2003; Folmer et al., 2003a) and secondly, the later the changes are made to the system the more expensive they are to implement. We recently presented a case study that consisted in quantifying the impact of the implementation of usability requirements changes on the architecture (Rafla et al., 2005). Some usability requirements may have some serious impacts on the architecture. The impact of the changes could have been reduced or even avoided if those usability requirements were defined and considered previously to the architectural design phase. It is to the best interest to determine as early as possible those architecturally sensitive usability requirements. Usability requirements should thus be well understood and articulated early in the development of a system, so that software designers can develop an architecture that will satisfy them. Few authors (Bass et al., 2001, 2003; Folmer et al., 2003a, 2004; Rafla et al., 2004, 2005) have investigated the relationship between usability and software architecture and little is known on how to identify and organize those usability requirements that are architecturally significant.

The Quality Attribute Workshop (QAW) (Barbacci et al., 2003), a method for eliciting and documenting quality attribute requirements early in the development process, provides mechanisms for capturing architecturally relevant requirements. It

helps uncover risks related to architectural decisions that might create future problems with regard to a quality attribute. The purpose of this paper is to propose a usability driven adaptation of the QAW (UQAW) and to show how this method can be modeled as an activity in a software development process. An exercise conducted to uncover the utility and the benefits of the UQAW, showed that participants found this method useful as it guided them in defining usability requirements that would severely impact the architecture if they were to be implemented in an existing application.

We begin by discussing recent work that investigated the link between usability and software architecture. QAW is highlighted and the usability driven adaptation of this method is then presented. We end by showing how practices of a software process can be adapted by using some of the QAW usability adopted methods.

5.2 Linking usability to software architecture

5.2.1 Usability driven software architecture patterns

Bass and his colleagues (2001, 2003) at the Software Engineering Institute (SEI) initiated the idea of linking usability to software architecture. They used a bottom-up approach by using field work observations of software development projects and generated scenarios that expressed a set of general usability issues that have potential architectural implications. They generated 27 scenarios and provided architectural patterns for implementing every scenario. For example, if a "help mechanism" is not devised when the software architecture is established, it will be very costly to incorporate afterwards. Note that they make no claim that their patterns are the only, nor even the best solution for every system. For example, Table 1, illustrating the undo scenario, describes the circumstances under which a user might need to undo an operation.

Table 5.1: Providing undo scenario (Bass and John 2003)

Providing undo
A user performs a command then no longer wants the effect of that operation. He/she should be able to return to the state the system was in before that operation was performed. It is also desirable to allow multi-level undo.

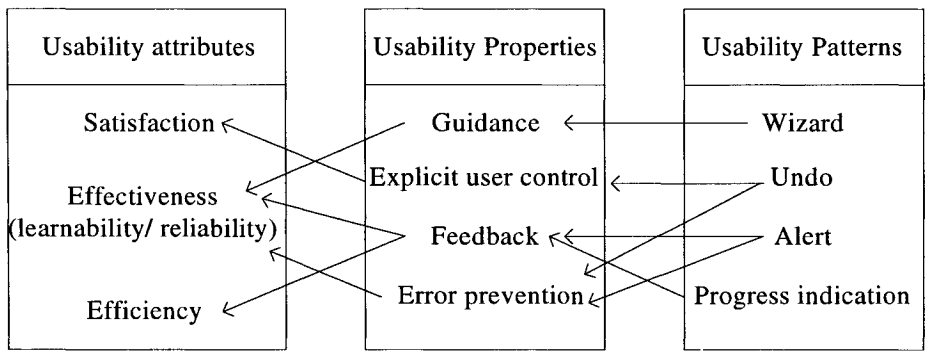
5.2.2 SofTware Architecture That supports USability (STATUS) project

In their investigation of the relationship between usability and software architecture, Folmer and collaborators (2003, 2004) started from the definition of usability, gradually refining this definition until they defined usability properties (similar to usability requirements) and finally linked those properties to architecturally sensitive usability patterns. The three defined layers are (Fig 1):

- *Attribute layer*: This first level of decomposition presents precise and measurable components of usability. Such attributes (ISO, 1998) are (i) learnability – how quickly and easily users can be productive with a system that is new to them; (ii) efficiency – the number of tasks per unit time that the user can perform; (iii) reliability – the error rate in using the system and the time it takes to recover from errors and finally (iv) satisfaction – the user's comfort highlighting positive attitudes towards the use of the system.
- *Properties layer*: The usability properties represent the design principles that seem to have a direct relationship on system usability. These properties can be used as requirements at the requirements gathering and analysis stage of a software development process (for example, providing feedback to the user, providing explicit user control, providing guidance, etc.). However, it is up to the usability analyst to guide the stakeholders in determining the set of properties they wish to see implemented.
- *Patterns layer*: The architecturally sensitive usability patterns provide a high level solution to a need specified by a usability property (Folmer and Bosch,

2003). Note that those patterns do not provide any specific software solution to be incorporated into a software architecture; they solely suggest some abstract mechanism that might be used to improve usability. It is up to the development team to assess whether implementing such a pattern will improve usability.

Figure 5.1: Folmer's framework (adapted from Folmer et al. 2003a)



Before we discuss how the architecturally relevant usability requirements can be dealt with proactively during the requirements elicitation, the next section highlights the original QAW, a framework that guides stakeholders in eliciting general quality requirements.

5.3 Quality Attribute Workshop: QAW

The SEI proposed the QAW a method that engages stakeholders early in the system development life cycle to discover the driving quality attributes of a software system. By providing a forum for a wide variety of stakeholders to gather very early in the development process, it leads to the identification of conflicting assumptions about quality requirements before the architecture has been created. The QAW process consists of the following eight steps:

1. QAW presentation and introduction: Representatives of the QAW team (quality analysts and senior analyst) describe the motivation for the QAW and explain each step of the method.

2. Business presentation: A representative of the stakeholder community discusses the system's business context and high-level requirements and constraints.
3. Architectural plan presentation: A technical stakeholder will present high-level system descriptions, drawings, or other artifacts that describe some of the system's technical details.
4. Scenario brainstorming: The facilitators initiate the brainstorming session in which stakeholders generate scenarios. Each stakeholder expresses a scenario representing his or her concerns with respect to the system.
5. Scenario consolidation: However, some of the elicited scenarios could be very similar in content and the comparable scenarios should be combined together. Quality analysts are next going to ask stakeholders to identify the scenarios that are very similar to the ones extracted during the brainstorming session. Similar scenarios are merged, as long as the people who proposed them agree and feel that their scenarios will not be undermined.
6. Scenario prioritization: At this point of the workshop, the scenarios gathered are classified and prioritized. The prioritization process is based on a voting procedure.
7. Scenario refinement: After the prioritization, the top four or five scenarios are refined in more detail.

Current usability specifications methods, such as user and task analysis, usability benchmarks and dialogue modeling (Ferre et al., 2001) have not addressed the elicitation of usability requirements that can impact an architecture. The next section discusses how the QAW can be adapted, by combining SEI and STATUS proposals, to guide software development organizations elicit and define architecturally sensitive usability requirements.

5.4 Usability driven adaptation of QAW: UQAW

5.4.1 The method

We propose to revise the QAW method by redefining and consolidating steps 4 and 5 in order to incorporate usability issues. Steps 1 to 3, 6 and 7 are left unchanged, as they are concerned with the presentation of the UQAW to stakeholders, the discussion of business requirements and the prioritization and refinement of scenarios. As users may have a vague understanding of usability, the new scenario brainstorming session should guide stakeholders in defining complete usability requirements. After that, similar scenarios should be merged together as some stakeholders may have identified similar usability requirements.

- *Step 4 - Scenario brainstorming*: This is a brainstorm exercise with the scope of identifying the usability requirements the system must support. Folmer's usability properties are initially presented to the stakeholders. Each stakeholder then expresses a requirement by underlining the usability property that will be fulfilled if that requirement is implemented. Finally, these requirements are formulated as scenarios.

  Some quality analysts are expected to have a deeper knowledge of the field of Human Computer Interaction (HCI). They are referred to as usability analysts and will assist stakeholders in eliciting usability requirements. The second step of the requirements generation phase consists of familiarizing the stakeholders with Bass's usability scenarios. Note that those scenarios are common to many interactive systems and are not related to the domain functionality of any system. The ones that stakeholders think they will inevitably increase the usability of the system are extracted.

- *Step 5 - Scenario consolidation*: Some of the elicited scenarios from the two previous steps could be very similar in content hence the comparable scenarios should be combined together. Usability analysts are next going to ask stakeholders to identify the scenarios that are very similar to the ones extracted

during the brainstorming session. Similar scenarios are merged, as long as the people who proposed them agree.

The next segment of this paper presents an example of how the UQAW method can be integrated into an existing software development process. We start by discussing the motivation behind this work.

5.4.2 Integration of UQAW into a software engineering process

5.4.2.1 Motivation

It is widely accepted that an integrated and collaborative software development process is imperative for efficient and productive interaction between software engineers and HCI practitioners (Seffah et al., 2005). All research that investigated the integration of usability activities into a software development process (Krutchen, 2001; Constantine et al., 2003; Ferre, 2003; Sousa and Furtado, 2003) perceived usability as being a function of the interface and lacked to mention the significance of software architectures in developing usable systems. John and her colleagues (2003), in a comparison of the *ISO 13407 Human-centred design processes for interactive systems* (1999) with the *Rational Unified Process: Best practices for software development teams* (2001) emphasize on the fact that the RUP is considered architecture centric and does not take into account user concerns, while ISO 13407 does not discuss software architecture as it neglects the importance of software architectures in fulfilling usability requirements.

In response, this work will show how the Unified Process for Education (UPEDU) (Robillard et al., 2003), an academic customization of the Rational Unified Process (RUP), can be adapted to have usability activities inserted into it. This software engineering process does not take into account architecturally sensitive usability requirements to construct the initial draft of the architecture. Since usability and software architecture are interdependent, how can the usability and software

engineering development processes be combined in order to guarantee that the constructed architecture would have usability aspects incorporated? A proposed solution consists in adapting UPEDU in order to have usability activities inserted into it. This has the aim of assisting developers in defining usability requirements that can impact an architecture and providing software designers with a solid basis for implementing usable systems. This effort consists in introducing the UQAW activity early in the software process, appending new roles and highlighting artifacts to be produced by them while performing this new activity.

UPEDU's characteristics are presented in the following section while our adaptation of this process is discussed afterwards.

5.4.2.2 UPEDU's decomposition

UPEDU is considered to be architecture-centric, meaning that development focuses on the elaboration of a software architecture as it provides the appropriate level of abstraction to support the design and understanding of complex systems. This process is decomposed into four major engineering disciplines: the requirements engineering discipline, the analysis & design discipline, the implementation discipline, and the test discipline. The requirements engineering discipline focuses on the elicitation of software requirements defined by the system analysts and from the identified needs of the users. Using the elicited requirements, the designers create the software architecture, components, component interfaces, and data elements during the analysis and design discipline. The implementation discipline is concerned with the programming of the system by developing components, and incrementally integrating them and producing working prototypes, while the test discipline evaluates the quality of the product, beginning early in the life cycle throughout the process until the evaluation of the final product delivered to the users.

A discipline is made of practices that are the collaboration between abstract active entities called *roles*, which perform operations called a*ctivities* on concrete, tangible entities called *artifacts*. It is worth mentioning that a role can either be played by an

individual or a small group of individuals. Activities on the other hand are a piece of work executed by one role. An artifact is any piece of information or physical entity produced or used by subsequent activities of the process. Examples of artifacts include models, plans, code, documentation, and so on. Figure 2 depicts the fundamental conceptual model.

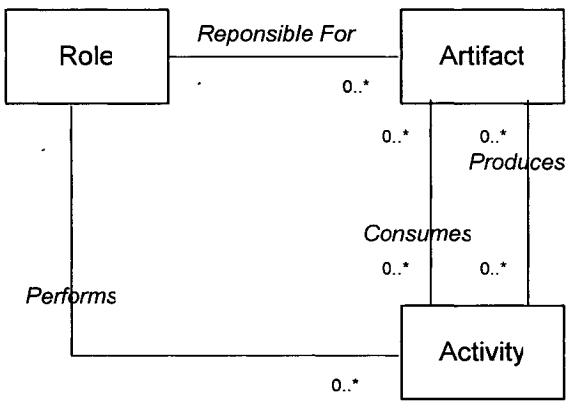


Figure 5.2: The conceptual model of UPEDU

5.4.2.3 Modeling UQAW as an activity in the requirements discipline

The UPEDU requirements discipline is concerned with understanding the proposed solution in order to define the software system to implement it, which is mainly based on comprehending users' needs. It consists of five activities carried out by the system analyst: eliciting the stakeholder request, finding actors and use cases which are essential to understanding the proposed solution, structuring and detailing the use cases responsible for defining the system, and finally reviewing the requirements (Fig 3). The gathering of usability requirements should be conducted in parallel with the eliciting the stakeholder request. Hence, the *understand the problem* workflow detail is the concerned workflow detail. The new requirements discipline would contain six activities and a new role, the usability analyst who is responsible for incorporating the user-centered methodology and guiding stakeholders in eliciting their usability requirements. Working with the quality analyst, the usability analyst is in charge of producing the supplementary specifications document that embodies other important

quality requirements, a complement to the use-case model. The supplementary specifications and the use-case model together capture a complete set of the system's functional and quality requirements.

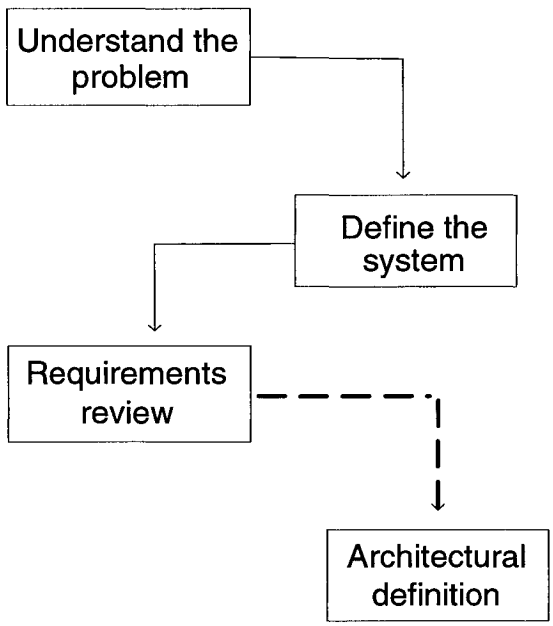


Figure 5.3: UPEDU's requirements workflow

UPEDU provides well-defined activities and guidelines for holding a requirements workshop as a means of eliciting stakeholder requests. Those requests are embodied in the use-cases and constitute solely the functionality of the system. These requirements specify primarily the system's functionality and are insufficient to specify an architecture that will address the system's usability. The proposed usability driven adaptation of the QAW, the UQAW is modeled as an activity in the "Understanding the Problem" sub-discipline of the requirements phase (Fig 4). Usability analysts work in close collaboration with stakeholders to identify possible problems from a usability perspective that might occur in the system. The result of the elicitation process is a list of needs that are described textually using scenarios, and that have been assigned a priority level. There is no explicit UPEDU artifact that corresponds to UQAW scenarios, but the information they contain about usability can be captured in the

supplementary specifications. The UQAW fills a need in UPEDU by providing an explicit method for gathering usability scenarios from stakeholders. The scenarios would be used as architecturally significant requirement when defining the candidate architecture.

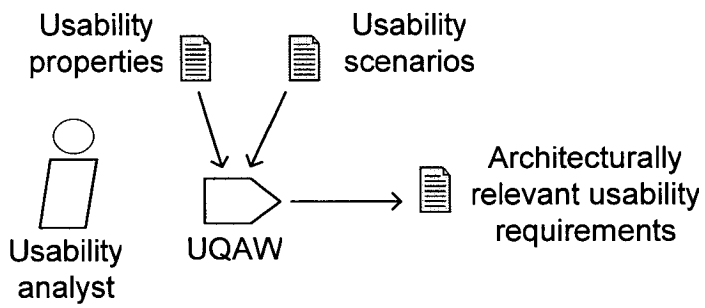


Figure 5.4: Extra activity in "understand the problem" workflow detail

Now that the UQAW and its integration into an architecture-centric software development process have been presented, this next section describes an exercise that was designed to assess the utility and benefit of the usability adaptation of QAW.

5.5 Validation of UQAW

5.5.1 Scope

It is our conjecture that some Folmer's usability properties are very high-level and sometimes difficult to grasp by average users. Providing more precise usability scenarios to stakeholders can guide them in defining better usability requirements. We need to understand the effect of providing the usability scenarios after the usability properties to ensure that all possible usability requirements are identified. More specifically, the application of the method aims to extract the following observations.

- Completeness of the set of usability properties:
  - Did the participants identify usability requirements that weren't covered in the usability properties?

    - Did participants experience difficulty understanding them?
- Completeness of the list of usability scenarios:
    - Did the participants identify usability requirements that weren't considered by the usability scenarios?
    - Did participants experience difficulty selecting the ones that are relevant?
- Was it helpful to provide the usability scenarios after familiarizing the participants with the usability properties?

5.5.2 Description of the exercise

The aim of this exercise is to ask participants to identify and elicit usability requirements for an existing system implemented by a team of software engineering students at Ecole Polytechnique de Montréal. This experiment provides valuable comments and feedback regarding the proposed adaptation of the QAW. The system, hereafter called Replan, is a web-based meeting management system aimed at organizers of meetings where the geographic dispersion makes scheduling difficult. This system allows meeting coordinators to send availability requests to a set of individuals, so that each of them can specify personal availability periods. The set of availability periods is then graphically displayed using a calendar tool to enable a coordinator to visualize the relevant information at a glance, making the scheduling decisions easier

5.5.3 Plan of the exercise

Six graduate computer engineering students from the Ecole participated in this exercise. Three of those participants had successfully taken several courses in human-computer-interaction, with one of those courses in Web usability. They are referred to as the *expert users* team. The other remaining participants are called the *average users*

team. The reason we grouped the participants in different teams according to their understanding of web usability is to determine whether the user's knowledge has an effect on the utility of the UQAW.

Before the actual exercise was conducted, participants received the functional requirements of the system and were asked to study them to attain a comprehensive knowledge of the system. This corresponds to steps 1 to 3 of the original QAW method.

The participants were then handed a document containing all nine usability properties identified by Folmer and his team, with the following aspects listed for each property. (Table 2 illustrates the *providing feedback* property).

- **Name:** the name of the property.
- **Intent:** a short statement explaining the property.
- **Usability attributed affected:** which usability attributes are affected (a "+" denotes a positive effect while a "-" denotes a negative one).
- **Example:** an illustration of the property.

Table 5.2: Providing feedback property (Folmer et al., 2003)

Providing feedback	
Intent:	The system should provide feedback to the user. He/she should be informed of what is going on, in other words what the system is doing.
Usability attributes affected:	Efficiency (+): users do not have to wonder what the system is doing. Learnability (+): users know what the system is doing.
Example:	Progress indication during a file download.

After having thoroughly examined the list of usability properties, stakeholders formulate usability requirements and associate them to usability properties. More specifically, they were asked to instantiate the properties in Replan. For example, one of the participants identified the need for Replan to provide a mechanism for end-users to retrieve forgotten passwords without external assistance. This usability requirement can

be associated to the explicit user control property. Thirdly, after having identified their requirements, Bass's list of 27 usability scenarios was provided to them. They were asked to select the ones that would apply to Replan.

5.5.4 Noted observations

All elicited usability requirements were examined to determine if, in fact, the participants were successful in incorporating the usability property in the formulated usability requirements. The following observations were noted:

- The usability properties correspond to very high-level usability requirements and their authors (Folmer et al., 2003) state that the more properties are implemented, the greater level of usability the system will attain.
    - The *expert users* team was able to elicit usability requirements that fulfill six of the nine the properties.
    - The *average users* team was able to elicit usability requirements that could be associated to three of the nine properties. All three participants did not provide requirements that covered the following usability properties, *consistency*, *guidance*, *minimize cognitive load*, *explicit user control* and *natural mapping*. This suggests that this set of usability properties are not clearly defined as participants experienced difficulty defining a usability requirement that fulfilled those properties.
- Only one participant from the *expert users* team was successful in identifying a requirement that would enable the fulfillment of the *explicit user control* property. This can be explained with the following observations:
    - It's the least explained property, as it does not contain enough information to guide users in identifying clear requirements. It is also the only property that affects solely the satisfaction usability

attribute. It's not easy to identify requirements that affect this usability attribute, as it is a psychological measurement of the user's comfort using an application and isn't easily understood by average users.

  - That participant the only one that has had a brief working experience with Replan.

- All six participants found the examples provided with the properties helpful as it guided them in the definition of usability requirements. However, they point out that more examples should be presented, categorized by the type of application they belong to. For example, if examples for web applications were provided, participants would have been more at ease defining usability requirements. We observe that the *accessibility* usability property presented an example that relates to web applications. This example mentions that the Cascading Style Sheets (CSS) standard allows developers to make specific style rules for web layout. This example proved useful, as it is the only property to have been covered by all participants.

Next, we explored the list of usability scenarios that each participant selected and the following observations were reported:

- The *expert users* team selected 21 scenarios compared to 14 for the *average users* team. Experienced participants were comfortable with the list of scenarios and were able to determine the relevance of some of the scenarios to Replan. On the other hand, the *average users* team found some scenarios rather confusing (e.g. *supporting comprehensive searching* and *operating consistently across views*) while others depicted the same usability aspect (e.g. *supporting visualization* and *making views accessible*) resulting in their non-inclusion.
- Was it helpful to provide the usability scenarios after familiarizing the participants with the usability properties?

- Two participants from the *average users* team did not correctly formulate a requirement for the *error management* property. However, they both selected the *checking for correctness* scenario, hence ensuring that an error correction/prevention mechanism is implemented in the final system. The *checking for correctness* scenario guided participants in asking for error correction to be enforced even though they failed to define a requirement for the *error management* property.
- Three participants (two from the *average users* team and one from the *expert users* team) did not correctly formulate a requirement for the *guidance* property. However, only the participant belonging to the *expert users* team selected the *providing good help* scenario and asserted that this scenario would help fulfill the *guidance* property and that it is important for Web applications. On the other hand, the three remaining participants (two of them are *expert users*) asked for a help system and/or tutorial even before seeing the *providing good help* scenario. This leads us to believe that the list of usability scenarios should be available during requirements elicitation to ensure that important usability concerns are not left out as some usability properties are weakly articulated and also since average users have an unclear understanding of usability requirements.
- Four participants (three from the *average users* team and one from the *expert users* team did not indicate that Replan should provide support for internationalization as part of the *accessibility* property. They specified that possible considerations for Personal Digital Assistant (PDA) and mobile devices should be supported. This actually depicts the second major decomposition of the *accessibility* property. Moreover, they all selected afterwards the *supporting international use* scenario hence ensuring that Replan is global in

order to reach the widest possible audience. On the other hand, the remaining two participants (*expert users*) were successful in defining the internationalization requirement as part of the *accessibility* property as well as choosing the *supporting international use* scenario. Here too, the list of usability scenarios is crucial during the gathering of requirements as the *average users* group when provided with the usability properties, failed to state that internationalization of Replan is an important usability requirement.

- Only one participant from the *expert users* team was able to identify a requirement that consisted in satisfying the *consistency* usability property. That participant asserts that the system should have the same visual layout for all operations as well as similar colors and forms for icons. After examining the list of usability scenarios, that participant states that this corresponds to the modifying visual interfaces scenario.

A brief questionnaire was handed out to obtain the participants' feedback regarding the UQAW method. Four of the six participants found that the usability properties provided a solid basis in the identification of their usability needs and guided them in formulating complete requirements. The other two participants (both from *expert users* team) found the properties high-level and more explanations and/or examples should have been provided. Moreover, five participants found that the usability scenarios concretely covered a great number of important usability requirements. They found them to be rather precise and that it wasn't difficult to see their relevance to Replan. Overall, all participants found the idea of providing the usability properties before the scenarios very helpful as it allowed them to acquire a general high level idea of user needs before eliciting their own requirements. It helped average users gain additional knowledge of the usability aspects that might impact Replan. The scenarios helped in the refinement of some of the already formulated requirements as well as discovering

new ones that might have been neglected when working with the usability properties.

The participants were asked if they have identified usability properties that weren't explicitly expressed as usability scenarios. The two participants who answered (both belonging to the *expert users* team) asserted that *minimize cognitive load* and the *natural mapping properties* were not captured in the scenarios. The participants were also asked if they have selected usability scenarios that weren't explicitly captured in the usability properties. The ones that were selected are: *using applications concurrently*, *operating consistently across views*, *making views accessible*, *evaluating the system* and *maintaining device independence*.

This experiment showed that the UQAW directed stakeholders in identifying and organizing architecturally relevant usability requirements. However, some improvements still need to be made. We notice that several participants experienced some difficulty understanding some of the usability properties, as the examples provided were very general. It would beneficial for the UQAW team to provide more examples as part of the property that are relevant to the system being developed.

5.6 From UQAW to architectural design

After having identified usability requirements, the next envisioned step is to identify specific mechanisms that might be incorporated into a software architecture to improve the usability of the final system (Fig 3). Software developers have no systematic way of incorporating usability requirements into their design. In other words, they need to know what particular elements of a software system need to be considered to implement a usability scenario. Recent research (Folmer and Bosch, 2003) have described an integrated set of "design solutions" that are considered to have a positive effect on the level of usability but that are difficult to retrofit in existing applications because these design solutions may require architectural support. The term used is *architecture-sensitive usability patterns* and refers to a mechanism that should be applied to the

design of the software architecture in order to address a need identified by a usability property at the requirements stage. The implementation of those patterns is basically a modification that may solve a specific usability problem in a specific context, and does not specify implementation details in terms of classes and objects. This is achieved by selecting appropriate usability patterns to satisfy the required usability properties.

After examining both usability properties and usability scenarios that were presented to the stakeholders in the requirements discipline, it became evident that the scenarios can be linked to usability properties. The scenarios provide a more tangible way of representing usability requirements and thus can condense the gap between usability properties and their analogous usability patterns. For example the *providing feedback* property, which discusses the need that the system should provide at every appropriate moment feedback to the user to keep them informed of what is going on, regroups the *predicting task duration*, *observing system state*, and *maintaining device independence* scenarios. Bass and his team (Bass et al., 2001) gave for each identified scenario an architectural pattern that provides a solution to implementing the scenario. Those architectural patterns reflect a possible solution, more specifically the implementation of a usability pattern

5.7 Concluding remarks

We nowadays understand that usability and software architecture are closely coupled. Even if the UI is properly designed, the system's usability could be compromised if certain usability improvement modifications cannot be easily accommodated by the architecture. The later the changes are made to the system, the more expensive they are to implement as certain architectural solutions may hamper usability requirements. Therefore, this type of quality requirement must be discovered and defined early in the development of a system to support the designer in constructing an architecture that will satisfy them. We investigated some avenues towards providing

means to guide stakeholders in eliciting their usability requirements that will ensure the proper alignment of those requirements with software architecture.

We discussed how the Quality Attribute Workshop (QAW), an effort by the SEI to define general quality requirements, can be adapted to provide a forum for stakeholders to share their concerns in terms of usability requirements early in the development. This workshop is modeled as an activity in the requirements discipline of a software engineering process. The usability analyst, a new role introduced in this discipline, helps stakeholders define and formalize their usability needs. The first step of the method consists in helping users define their own requirements by providing them with a list of usability properties, inspired from the work of Folmer. Users were asked to express their requirement by underlining which property will be fulfilled if that requirement was implemented. The usability analyst would guide stakeholders in formulating those requirements as scenarios. Since some users often have a vague comprehension of the own requirement, the second step consists in presenting them with already formulated usability scenarios from the work of Bass that have an impact on the architecture. Users were asked to extract the ones that they wished to see implemented. Finally, the third step is concerned with the identification of similar scenarios from the two previous steps. Similar scenarios are merged. An exercise designed to assess the benefit and utility of the UQAW activity was conducted. Participants found the usability properties useful as it lead them in articulating their usability requirements. Also, they found the usability scenarios detailed as they provided them with a concrete example of usability requirements.

Some work is still needed to ensure that this work will guide software engineers in architecting for better usability. The validation of the effectiveness of the usability driven adaptation of UPEDU, or any other software engineering process that integrates UQAW, needs to be explored. This has the aim of assessing the utility of the added UQAW activity from a software process perspective. Are software engineers at ease carrying out this new activity? What are the cognitive activities that need to be performed in order to guarantee that the usability requirements brainstorming session is

a success? Also supplementary research is needed to assert how to translate the elicited usability requirements from the UQAW into architectural solutions. Software architectural expertise is needed to carry out the usability-driven architectural activities that are introduced in the analysis and design discipline.

Acknowledgements

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) under grants A0141 and 121923-03. The authors would like to send their sincerest gratitude to the participants for partaking in the experiment and also for providing the details of their work.

DISCUSSION, CONCLUSION ET RECOMMANDATIONS

Suite à une étude approfondie de la littérature, seulement deux groupes de recherche ont étudié le lien qui existe entre l'architecture et l'utilisabilité. Ils ont défini des exigences d'utilisabilité qui ne sont pas satisfaites lors de la séparation de l'interface du système. Ces exigences nécessitent une sérieuse considération lors de l'élaboration de l'architecture. Aussi, ces groupes de recherche ont proposé des patrons d'architecture qui permettent l'implantation de ces requis d'utilisabilité. Dorénavant, certains experts du domaine de l'interaction homme-machine affirment que les deux approches proposées sont plus ou moins similaires, puisque certaines exigences identifiées se chevauchent. De plus, il reste beaucoup d'avenues à être explorées. Les aspects qui doivent être étudiés incluent :

- Proposer des méthodes qui permettent l'analyse et l'évaluation d'une architecture afin de vérifier si les exigences d'utilisabilité ont été incorporées avec succès. Ces méthodes auront principalement pour objectif de vérifier la présence des patrons d'architecture qui sont associés aux exigences d'utilisabilité.
- Les exigences d'utilisabilité définies par ces deux groupes de recherches sont-elles architecturalement sensibles? Quel impact architectural ces exigences vont-elles encourir si elles sont implantées dans une architecture existante ?
- Certaines exigences peuvent sérieusement avoir un impact sur l'architecture puisque cette dernière n'accommodera pas facilement les changements d'utilisabilité. Ces exigences ont dû être définies avant l'élaboration de l'architecture, plus précisément lors de l'analyse des besoins des intervenants. Malheureusement, aucune méthode n'existe qui permet de guider les intervenants dans la détermination des requis d'utilisabilité qui peuvent impacter l'architecture.

- Une fois qu'une telle méthode est conçue, il serait intéressant de modéliser cette méthode comme activité et d'identifier les rôles et artefacts qui sont associés à cette activité. Ceci a pour objectif de montrer aux organisations qui développent des logiciels comment intégrer cette méthode dans leurs processus existants.

Nous avons avancé la recherche en illustrant comment le SAAM (cf. chapitre 3) peut être adapté afin d'aider les concepteurs logiciels dans la conception d'architectures qui soutiennent l'utilisabilité. Notre application de SAAM, qui est basée sur les scénarios d'utilisabilité de Bass se compose de quatre étapes principales : (1) choisir l'ensemble de scénarios d'utilisabilité qui s'appliquent au système, (2) la description l'architecture, (3) l'évaluation des scénarios et (4) l'évaluation globale et la discussion. La première étape de la méthode consiste à extraire les scénarios qui sont applicables. On a proposé ensuite des solutions de conception, qui ne sont pas toutes nécessairement identiques aux patrons d'architecture définis, qui permettent l'implantation des scénarios choisis. Les solutions proposées ont été obtenues par un processus inductif qui garantit que ces solutions sont possibles, quoique pas nécessairement les seules. Après l'analyse de deux architectures distinctes du même système, notre étude montre que deux des six scénarios choisis pourraient être mis en application dans un système existant en encourant de sérieux changements architecturaux. Ces effets auraient pu être évités si ces scénarios d'utilisabilité avaient été considérés dans la phase de conception. Malheureusement, les processus centrés utilisateur, comme la norme ISO 13407 ne définissent pas d'architectures de logiciel et donc les exigences d'utilisabilité qui nécessitent une considération dans l'architecture ne pourraient pas vraiment être mis en application. Afin de pouvoir raisonner sur ces exigences d'utilisabilité, UPEDU, un processus centré architecture peut être adapté pour prendre en considération l'utilisabilité.

Nous avons ensuite montré comment l'atelier d'attribut de qualité (QAW), un effort par le SEI pour définir les requis de qualité générale, peut être adapté pour fournir un forum pour les intervenants pour partager leurs soucis en termes d'exigences d'utilisabilité. Cet atelier est modélisé comme une activité dans la discipline des requis de UPEDU. La première étape de la méthode consiste à guider les utilisateurs dans la définition de leurs exigences d'utilisabilité en leur présentant une liste de propriétés d'utilisabilité, inspirée du travail de Folmer. Les intervenants vont exprimer leur exigence en soulignant quelle propriété sera accomplie si ce requis est implanté. L'analyste de l'utilisabilité guidera les intervenants dans la formulation de ces exigences en scénarios. Puisque beaucoup d'utilisateurs ont une compréhension vague de l'utilisabilité, la deuxième étape consiste à présenter les scénarios déjà formulés de Bass. Les utilisateurs vont extraire ceux qu'ils souhaiteraient voir mis en application. Finalement, la troisième étape est concernée par l'identification des scénarios semblables des deux étapes précédentes. Les scénarios semblables sont fusionnés.

Une expérience empirique conçue pour évaluer l'utilité de l'activité d'UQAW a été effectuée. Les participants ont trouvé les propriétés d'utilisabilité utiles dans la mesure où elles les ont guidés dans la formulation de leurs exigences. En outre, ils ont trouvé les scénarios détaillés puisqu'ils leur fournissaient un exemple concret d'exigences d'utilisabilité.

La section qui suit, a pour objectif d'énoncer certaines limitations de nos travaux de recherche. Pour chacun des articles qui constituent le corps de ce mémoire, nous allons indiquer les aspects que nous n'avons pas nécessairement explorés. Ces aspects seront ensuite formulés comme futures avenues de recherches.

Au chapitre 4, nous avons montré que certains changements d'utilisabilité ont un sérieux impact sur l'architecture d'une application web. Afin de bien quantifier cet impact, les solutions de conception proposées ont été analysées dans deux architectures distinctes du même système. La raison pour laquelle nous avons seulement étudié

l'impact de ces changements sur deux architectures est due au fait que les autres systèmes développés par les étudiants, soit n'étaient pas opérationnels, soit certaines fonctionnalités avaient été oubliées.

De plus, cette quantification n'est pas tout à fait précise et il aurait été avantageux d'incorporer les solutions proposées dans un plus grand nombre d'architectures. De plus, nous pouvons affirmer que les impacts perçus sont spécifiques aux applications web. Cependant, aurions-nous identifié la même magnitude d'impact si les solutions proposées avaient été incorporées dans d'autres types d'applications ? Une récente étude (Rafla et al. 2004) montre que l'incorporation de certaines exigences d'utilisabilité dans l'architecture de GIMP, un système de traitement d'images à source ouvert a été possible sans requérir de sérieux changements à l'architecture. Cette conclusion est soutenue par l'hypothèse que les architectures des logiciels à source ouverte doivent être assez flexibles pour accommoder les nouvelles fonctionnalités qui sont ajoutées.

Au chapitre 5, nous avons proposé le UQAW (Usability Attribute Quality Workshop) pour permettre aux organisations développant des logiciels, d'identifier et d'organiser les requis d'utilisabilité qui requièrent une réflexion tôt dans le cycle de développement. Une expérimentation réalisée avec deux groupes de participants, des experts et des personnes ayant une connaissance moyenne en utilisabilité des applications web, a démontré que les participants n'ont pas éprouvé de la difficulté à exprimer des exigences d'utilisabilité. Aurions-nous obtenu ce même résultat encourageant si les participants recrutés pour cette étude n'avaient aucune connaissance ni expérience en utilisabilité ? Certaines exigences d'utilisabilité semblent complexes et ne sont pas très bien expliquées. On pourrait croire que des résultats aussi encourageants seront obtenus si les personnes qui administrent cet atelier ont une compréhension approfondie du domaine de l'utilisabilité et pourront compenser le manque de connaissance chez les intervenants.

Cette méthode a été ensuite intégrée dans UPEDU afin de fournir aux ingénieurs une base solide pour identifier les exigences d'utilisabilité. Malheureusement, aucune étude ne permet de mesurer l'effort de conception et les activités cognitives que les membres de l'équipe de développement devront faire afin d'accomplir cette activité. Aussi, cette méthode garantit-elle un système de meilleure utilisabilité ? Des tests d'utilisabilité effectués sur un produit fini après avoir employé cette méthode permettra de mesurer l'utilisabilité de l'application développée.

BIBLIOGRAPHIE

AYKIN, Nuray. 1999. "Internationalization and localization of web sites". *8th International Conference on Human-Computer Interaction.* Munich, Germany, pp. 1218-1222.

BASS, Len, JOHN, Bonnie E, KATES, Jesse. 2001. *Achieving usability through software architecture.* Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University.CMU/SEI-TR-2001-005.

BASS, Len, JOHN, Bonnie E. 2003. "Linking usability to software architecture patterns through general scenarios". *Journal of Systems and Software*, Elsevier, 66(3): 187-197.

BASTIEN, Christian. 1991. *Validation de critères ergonomiques d'interfaces utilisateurs.* Rapport recherche INRIA n° 4127, programme 3.

BARBACCI, Mario, ELLISON, Robert, LATTANZE, Anthony, STAFFORD Judith A., WEINSTOCK, Charles B., WOOD, William G., 2003. *Quality Attribute Workshop (QAW)*, 3rd ed. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University. CMU/SEI-2003-TR-016.

BENGTSSON, PerlOlof, BOSCH, Jan. 2001. "Assessing optimal software architecture maintainability. *5th European Conference on Software Maintainability and Reengineering.* Lisbon, Portugal, pp. 168-175.

BENNET, John, KARAT, John. 1994. "Facilitating effective HCI designs". *SIGCHI Conference on human factors in computing systems: celebrating interdependence*, Boston, MA, USA, pp 198-204.

BEVAN, Nigel. 1995. "Measuring Usability as Quality of Use". *Journal of Software Quality* (4): 115-140.

BOEHM, Barry, BROWN, John, KASPAR, Hans. 1978. "Characteristics of Software Quality". *TRW Series of Software Technology.*

CLEMENTS, Paul, KAZMAN, Rick, KLEIN, Mark. 2001. *Evaluating software architectures: methods and case studies.* Addison Wesley, MA.

CONSTANTINE, Larry, LOCKWOOD, Lucy. 1999. *Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design.* Addison-Wesley, New York, NY.

CONSTANTINE, Larry, BIDDLE, Robert, NOBLE James. 2003. Usage-centered design and software engineering: Models for integration. *International Conference on Software Engineering (ICSE)*, pp. 106-113

COUTAZ, Joëlle. 1987. "PAC: an Implementation Model for Dialog Design". *2nd IFIP Conference on Human-Computer Interaction (Interact)*, Stuttgart, Germany, pp. 421-436

DOBRICA, Liliana, NIEMELA, Eila. 2002. "A survey on software architecture analysis methods". *IEEE Transactions on Software Engineering*, 28(7): 683-654.

DUMAS, J., REDISH, G. 1993. A practical Guide to Usability Testing. New Jersey: Ablex Publishing Corp.

EASON, Ken. 1984. "Towards the experimental study of usability". *Behaviour and Information Technology* 3(2): 133-143.

FERRE, Xavier, JURSITO, Natalia, WINDL, Helmut, CONSTANTINE, Larry. 2001. "Usability basics for Software Developers". *IEEE Software* 18(1), January/February, pp. 22-29.

FERRE, Xavier, JURSITO, Natalia, MORENO, Anna M., SANCHEZ M. Isabel, 2003. "A Software Architectural view of Usability Patterns". *2nd Workshop on Software and Usability Cross-Pollination: The Role of Usability Patterns (Interact).* Zurich, Switzerland.

FERRE, Xavier. 2003. "Integration of usability techniques into the software development process. *ICSE Workshop on bridging the gaps between software engineering and human computer interaction.* Portland, USA, pp. 28-35.

FOLMER, Eelke, BOSCH, Jan. 2003. "Usability patterns in software architecture". *10th International Conference on HCI.* Crete, Greece, pp. 93-97.

FOLMER, Eelke, VAN GURP, Jilles, BOSCH, Jan. 2003a. "A framework for capturing the relationship between usability and software architecture". *Software Process – Improvement and Practice: Special Issue on Bridging the Process and Practice Gaps between Software Engineering and Human-Computer Interaction*, 8(2): 67-87.

FOLMER, Eelke, VAN GURP, Jilles, BOSCH, Jan. 2003b. "Scenario based assessment of software architecture usability". *ICSE Workshop on bridging the gaps between software engineering and human computer interaction*, Portland, USA, pp. 61-68

FOLMER, Eelke, BOSCH, Jan. 2004. "Architecting for usability: a survey". *Journal of Systems and Software*, Elsevier, 70(1-2): 61-78.

GARVIN, David. 1984. "What does product quality really mean?" *Sloan Management Review*, pp. 25–45.

GOLDBERG, Adele, ROBSON, David. 1983. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, MA

GULLIKSEN, Jan, LANTZ, Ann, BOIVIE, Inger. 1999. "User centered design: problems and possibilities". A summary of PDC and CSCW workshop. 31(2): 25-35

HAGGANDER, Daniel, BENGTSSON, PerlOlof, BOSCH Jan, Lundberg, Lars. 1999. "Maintainability myth cases: performance problems in parallel applications". *IASTED 3rd International Conference on Software Engineering and Applications.* Arizona, USA, pp. 288-294.

HIX, Deborah, Hex, Rex. 1993. *Developing user interfaces: ensuring usability through product and process.* Wiley, 381p.

INDERJEET, Singh, STEARNS, Beth, JOHNSON Mark. 2002. *Designing Enterprise Applications with the J2EE Platform.* 2ed, Addison-Wesley, CO.

ISO/IEC 9241-11. Ergonomic requirements for office work with visual display terminals (VDT) s - Part 11 Guidance on usability, ISO/IEC 9241-11: 1998 (E).

ISO/IEC 13407. Human-Centered Design Processes for Interactive Systems, ISO/IEC 13407: 1999.

ISO/IEC 9126. software engineering - product quality. Technical report, International Standards Organisation. 2003.

JOHN, Bonnie E, BASS, Len, ADAMS. Rob J. 2003. "Communication across the HCI/SE divide: ISO 13407 and the Rational Unified Process". *10th International Conference on HCI.* Crete, Greece.

KAZMAN, Rick, ABOWD, Gregory, BASS, Len, WEBB, Mike, 1994. "SAAM: A method for analyzing the properties of software architectures". *16th International Conference on Software Engineering*, Sorrento, Italy, pp. 81-90.

KAZMAN, Rick, ABOWD, Gregory, BASS, Len, CLEMENTS, Paul. 1996. "Scenario based analysis of software architecture". *IEEE software,* 13 (6): 47-56

KITCHENHAM, Barbara, PFLEEGER, Shari. 1996. "Software quality: The elusive target". *IEEE Software*, 1:12–21.

KRASNER, Glenn, POPE. Stephen. 1988. "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80". *Journal of Object-Oriented Programming*, 1(3):26-49.

KRUTCHEN, Philippe. 2000. *The Rational Unified Process: An introduction*, 2ed. New Jersey. Addison-Wesley.

KRUTCHEN, Philippe, AHLQVIST, Stefan, BYLUND, Stefan. 2001. "User Interface design in the Rational Unified Process". *Object modeling and user interface design.* Addison-Wesley: Boston, pp.131-196.

LI, Wei, HENRY, Sallie, 1993. "OO metrics that predict maintainability". *Journal of Systems and Software*, Elsevier, 23(2): 111-122.

MYERS, Brad, ROSSON, Mary Beth. 1992. "Survey on user interface programming". *ACM CHI'92 Conference.* Monterey, CA, pp. 195-205

NIELSEN, Jakob, MOLICH, Rolf. 1990. "Heuristic evaluation of user interfaces". *Conference on Computer Human Interaction*, Seattle, USA. ACM, pp.349-256.

NIELSEN, J. 1992. Finding usability problems through heuristic evaluation. *ACM CHI'92 Conference*. Monterey, CA, pp. 373-380.

NIELSEN, Jakob, LANDAUER, Thomas K. 1993. "A mathematical model of the finding of usability problems". *ACM INTERCHI'93 Conference*. Amsterdam, The Netherlands, pp. 206-213.

NIELSEN, Jakob, 1993. *Usability engineering*. Boston, MA: AP Professional.

NIELSEN, Jakob, 1994. « Enhancing the explanatory power of usability heuristics". *ACM Computer Human Interaction (CHI 94) Conference*. Boston, MA, pp. 152-158.

NIELSEN, Jakob. 2000. *Designing Web Usability: The Practice of Simplicity*. Indianapolis, IN, New Riders Publishing.

NIELSEN, Jakob, 2002. Alertbox December 2002: http://www.useit.com/alertbox/20021223.html (accédée le 7 Février 2005)

NORMAN, Donald, 1984. *Cognitive engineering*. In user centered system design. Lawrence Erlbaum Associates, pp.31-61.

MCCALL, James, WALTERS, Gene. 1977. *Factors in software quality*. Technical report US Rome Air Development Center Reports.
PFAFF, Günther. 1985. User interface management systems. New York: Springer.

RAFLA, Tamer, OKETOKOUN, Rafiou, WIKLIK, Artur, DESMARAIS, Michel, ROBILLARD, Pierre-N. 2004. "Accommodating usability-driven changes in existing software architecture". *IASTED 8th International Conference on Software Engineering and Applications*. Cambridge, USA, pp. 150-154.

RAFLA, Tamer, ROBILLARD, Pierre N, DESMARAIS, Michel. 2005. "Investigating the impact of usability on software architecture through scenarios: a case study on web systems". *Journal of Systems and Software*, Elsevier. In Press

RAFLA, Tamer, ROBILLARD, Pierre N., DESMARAIS, Michel. 2005. "A method to elicit architecturally sensitive usability requirements: its integration into a software development process". *Submitted to the Software Quality Journal, Springer.*

RHEAUME, Jacques. 1991. *Hypermédias et stratégies pédagogiques.* Hypermédias et Apprentissages, actes des premières journées scientifiques, INRP.

ROBILLARD, Pierre N., KRUTCHEN, Philippe, D'ASTOUS Patrick. 2003. *Software engineering process with UPEDU.* Boston: Addison Wesley. 342p.

RUBIN, Jeffrey, 1994. *Handbook of Usability Testing, How to Plan, Design, and Conduct Effective Tests.* New York: John Wiley & Sons, Inc

SCAPIN, Dominique. 1986. *Guide ergonomique de conception des interfaces Homme/Machine.* Rapport recherche INRIA n° 77.

SCHMUCKER, Kurt. 1986. Object-*Oriented Programming for the Macintosh.* Hayden, Hasbrouck Heights, New Jersey.

SEFFAH, Ahmed, GULLIKSEN, Jan, DESMARAIS, Michel (Eds.). 2005. *Human-Centered Software Engineering: Integrating Usability in the Development Process.* Springer Publishing.

SENACH, Bernard. 1990. *Evaluation ergonomique des interfaces home-machine: une revue de la littérature.* Rapport recherche INRIA n° 1180, programme 8, Communication Homme-Machine.

SHACKEL, Brian. 1981. "The concept of usability". *IBM software and information usability symposium*, New York, USA, pp 1-30.

SHACKEL, Brian.1986. "Ergonomics in design for usability". *Human Computer Interaction (HCI) conference*, Cambridge, UK. Cambridge University Press.

SHACKEL, Brian, RICHARDSON, Simon (Eds.). 1991. *Human factors for informatics usability.* Cambridge, UK: Cambridge University Press

SHNEIDERMAN, Ben. 1998. *Designing the user interface: Strategies for effective human-computer interaction.* 3rd ed. Reading, MA: *Addison-Wesley*

SOUSA, Kenia, FURTADO, Elizabeth. 2003. "A approach to integrate HCI and SE in requirements engineering". *Closing the gaps: Software Engineering and Human-Computer-Interaction (Interact).* Zurich, Switzerland, pp. 81-88.

SOUSA, Kenia, FURTADO, Elizabeth. 2003. "RUPi – A Unified Process that Integrates Human-Computer Interaction and Software Engineering". *International Conference on Software Engineering (ICSE).* Portland, USA, pp. 41-48.

The Status Project. 2002. http://www.ls.fi.upm.es/status/ (accédée le 15 Juin 2005)
UPEDU website companion. http://www.upedu.org. (accédée le 15 Juin 2005)

ANNEXE A

USABILITY PROPERTIES

Providing feedback	
Intent:	The system should provide at every moment feedback to the user in which case he or she are informed of what is going on, i.e. what the system is doing.
Usability attributes affected:	*Efficiency* (+): users do not have to wonder what the system is doing. *Learnability* (+): users know what the system is doing.
Example:	Progress indication during a file download.

Error management	
Intent:	The system should prove a way to manage user errors. This can be done in two ways: • By preventing errors to happen, so users can make no or less mistakes. • By providing an error mechanism, so that errors made by users can be corrected.
Usability attributes affected:	*Reliability* (+): error management increases reliability because users make fewer mistakes. *Efficiency* (+): efficiency is increased because it takes less time to recover from errors.
Example:	Red underline for a syntax error in Eclipse (a popular Java development environment)

Consistency	
Intent:	Users should not have to wonder whether different words, situations, or actions mean the same thing. It is regarded as an essential design principle that consistency should be used within applications. It can be provided in different ways: • Visual consistency: user interface elements should be consistent as well in aspect and structure • Functional consistency: the way of

	performing different tasks across the system should be consistent, but also with other similar systems, and even between different kinds of applications in the same system. • Evolutionary consistency: in the case of a software product family, consistency over the products in the family is typically considered an important aspect.
Usability attributes affected:	*Learnability* (+): consistency makes learning easier because concepts and actions have to be learned only once. *Reliability* (+): visual consistency increases perceived stability, which increases user confidence in different new environments.
Example:	Most applications for MS Windows conform to standards and conventions with respect to e.g. menu layout and key-bindings.

Guidance	
Intent:	In order to help the user understand and use the system, the system should provide informative, easy to use, and relevant guidance and support in the application as well as in the user manual.
Usability attributes affected:	*Learnability* (+): guidance informs the user at once which steps or actions will need to be taken and where the user currently is, which increases learnability. *Efficiency* (-): guidance may decrease efficiency as users are forced to follow guidance. Reliability (+): when users are forced to follow a sequence of tasks, users are less likely to miss important things and will hence make fewer errors.
Example:	ArgoUML, a popular UML modeling tool auto generates a to-do list based on lacking information in models under construction.

Minimize cognitive load	
Intent:	Humans have cognitive limitations; therefore we bear in mind these limitations. For example, presenting more than seven items on the screen is considered an overload of information.
Usability attributes affected:	*Reliability* (+): objects or functions not of the interest of users do not distract them, and users are less

	likely to make errors. *Efficiency* (+): minimizing the cognitive load may increase efficiency since users are not distracted by actions that are not important to them. *Efficiency* (-): this is not entirely true if the user is an expert using the application.
Example:	The auto hide feature in office applications

Explicit user control	
Intent:	Direct manipulation should be supported; e.g.; the user should get the impression that he is "in control" of the application.
Usability attributes affected:	*Satisfaction* (+): interaction is more rewarding if users feel that they directly influence the objects instead of just giving the system instructions to act.
Example:	The cancel button when copying a large file allows users to interrupt the operation

Natural mapping	
Intent:	The system should provide a clear relationship between what the user wants to do and the mechanism doing it. This property can be structured as follows: • Predictability: the system should be predictable. • Semiotic significance: systems should be semiotically significant. (Semiology is the study of signs, symbols, and signification) • Ease of navigation: it should be obvious to the user how to navigate the system.
Usability attributes affected:	*Learnability* (+): if the system provided a clear relationship between that users want to do and the mechanism for doing it, users have less trouble learning something that is already familiar to them. *Efficiency* (+): a clear relationship between what needs to be done and how to do it may increase efficiency. *Reliability* (+): a clear relationship between what and how minimizes the number of errors made accomplishing a task.
Example:	The drag and drop recycle bin on the desktop is an easy to remember metaphor.

Accessibility	
Intent:	Systems should be accessible in everyway that is required. Such property might be decomposed as follows: • Disabilities: systems should provide support for users who are disabled. • Multi-channeling: the system should be able to support access via various media. (Browse a website via a phone or also being able to auditory browse a website) • Internationalization: systems should provide support for internationalization, because users are more familiar with their own language, date format, etc.
Usability attributes affected:	*Satisfaction* (+): accessibility may increase satisfaction by allowing the use of the system adapted to their familiar context *Learnability* (+): learnability may be improved for internationalization because users are more familiar with their own language, etc.
Example:	The W3C Cascading Style Sheet (CSS) standard supports multi-channeling by allowing developers to make specific styles rules for web and printer layouts.

Adaptability	
Intent:	The system should be able to satisfy user needs when the context changes or also adapt to changes in the user. Such property can be decomposed as follows: • User experience: ability to adapt to changes in the user's level of experience. • Customization: ability to provide certain customized services. • System memorability: capacity of the system to remember past details of the user-computer interaction.
Usability attributes affected:	*Satisfaction* (+): satisfaction may be increased because users can express their personal likes and preferences. *Efficiency* (+): the system will adapt to the skills, preferences or knowledge of the user, which may increase user's efficiency.
Example:	Users can apply a skin they have downloaded to the interface of the Winamp application.

ANNEXE B

USABILITY SCENARIOS

Aggregating data. A user may want to perform one or more actions on more than one object. Systems should allow users to select and act upon arbitrary combinations of data.

Aggregating commands. A user wishes to perform a multi-step procedure repetitively. Systems should provide a batch or macro capability to allow users to aggregate commands.

Cancelling commands. A user invokes an operation, then no longer wants the operation to be performed. Systems should allow users to cancel operations.

Using applications concurrently. A user may want to work with arbitrary combinations of applications concurrently. These applications may interfere with each other. Systems should ensure that users can employ multiple applications concurrently without conflict.

Checking for correctness. A user may make an error that he or she does not notice. However, human error is frequently circumscribed by the structure of the system; the nature of the task at hand, and by predictable perceptual, cognitive, and motor limitations. Depending on context, error correction can be enforced directly or suggested through system prompts.

Maintaining device independence. A user attempts to install a new device. The device may conflict with other devices already present in the system. Alternatively, the device may not function in certain specific applications. When device conflicts occur, the

system should provide the information necessary to either solve the problem or seek assistance

Evaluating the system. A system designer or administrator may be unable to test a system for robustness, correctness, or usability in a systematic fashion. Systems should include test points and data gathering capabilities to facilitate evaluation.

Recovering from failure. A system may suddenly stop functioning while a user is working. Users should be provided with the means to reduce the amount of work lost from system failures.

Retrieving forgotten passwords. A user may forget a password. Retrieving and/or changing it may be difficult or may cause lapses in security. Systems should provide alternative, secure strategies to grant users access.

Providing good help. A user needs help. The user may find, however, that a system's help procedures do not adapt adequately to the context. Help content may also lack the depth of information required to address the user's problem. Help procedures should be context dependent and sufficiently complete to assist users in solving problems.

Reusing information. A user may wish to move data from one part of a system to another. Users should be provided with automatic (e.g., data propagation) or manual (e.g., cut and paste) data transports between different parts of a system.

Supporting international use. A user may want to configure an application to communicate in his or her language or according to the norms of his or her culture. Systems should be easily configurable for deployment in multiple cultures.

Leveraging human knowledge. People use what they already know when approaching new situations. Such situations may include using new applications on a familiar platform, a new version of a familiar application, or a new product in an established product line. System designers should strive to develop upgrades that leverage users' knowledge of prior systems and allow them to move quickly and efficiently to the new system.

Modifying interfaces. Iterative design is the lifeblood of current software development practice, yet a system developer may find it prohibitively difficult to change the UI of an application to reflect new functions and/or new presentation desires. System designers should ensure that their UIs can be easily modified.

Supporting multiple activities. Users often need to work on multiple tasks more or less simultaneously (e.g., check mail and write a paper). A system or its applications should allow the user to switch quickly back and forth between these tasks.

Navigating within a single view. A user may want to navigate from data visible on-screen to data not currently displayed. If the system takes too long to display the new data or if the user must execute a cumbersome command sequence to arrive at her or his destination, the user's time will be wasted. System designers should strive to ensure that users can navigate within a view easily and attempt to keep wait times reasonably short.

Observing system state. A user may not be presented with the system state data necessary to operate the system (e.g., uninformative error messages, no file size given for folders). Alternatively, the system state may be presented in a way that violates human tolerances (e.g., is presented too quickly for people to read. See: Working at the user's pace). The system state may also be presented in an unclear fashion, thereby confusing the user. System designers should account for human needs and capabilities when deciding what aspects of system state to display and how to present them.

Working at the user's pace. A system might not accommodate a user's pace in performing an operation. This may make the user feel hurried or frustrated. Systems should account for human needs and capabilities when pacing the stages in an interaction.
Systems should also allow users to adjust this pace as needed.

Predicting task duration. A user may want to work on another task while a system completes a long running operation. If systems do not provide expected task durations, users will be unable to make informed decisions about what to do while the computer ''works.'' Thus, systems should present expected task durations.

Supporting comprehensive searching. A user wants to search some files or some aspects of those files for various types of content. Search capabilities may be inconsistent across different systems and media, thereby limiting the user's opportunity to work. Systems should allow users to search data in a comprehensive and consistent manner by relevant criteria.

Supporting undo. A user performs an operation, then no longer wants the effect of that operation. The system should allow the user to return to the state before that operation was performed. Furthermore, it is desirable that the user then be able to undo the prior operation (multi-level undo).

Working in an unfamiliar context. A user needs to work on a problem in a different context. Discrepancies between this new context and the one the user is accustomed to may interfere with the ability to work. Systems should provide a novice (verbose) interface to offer guidance to users operating in unfamiliar contexts.

Verifying resources. An application may fail to verify that necessary resources exist before beginning an operation. This failure may cause errors to occur unexpectedly

during execution. Applications should verify that all necessary resources are available before beginning an operation.

Operating consistently across views. A user may become confused by functional deviations between different views of the same data. Commands that had been available in one view may become unavailable in another or may require different access methods.

Systems should make commands available based on the type and content of a user's data, rather than the current view of that data, as long as those operations make sense in the current view.

Making views accessible. Users often want to see data from other viewpoints. If certain views become unavailable in certain modes of operation, or if switching between views is cumbersome, the user's ability to gain insight through multiple perspectives will be constrained.

Supporting visualization. A user wishes to see data from a different viewpoint. Systems should provide a reasonable set of task-related views to enhance users' ability to gain additional insight while solving problems.

Supporting personalization. A user wants to work in a particular configuration of features that the system provides. The user may want this configuration to persist over multiple uses of the system (as opposed to having to set it up each time). Systems should enable a user to specify their preferences for features and provide the possibility for these preferences to endure.

ANNEXE C

ARTICLE DE CONFERENCE

ACCOMODATING USABILITY-DRIVEN CHANGES IN EXISITING SOFTWARE ARCHITECTURE

Tamer Rafla, Rafiou Oketokoun, Artur Wiklik, Michel Desmarais, Pierre N. Robillard
8th IASTED International Conference on Software Engineering and Applications (SEA). November 2004, Cambridge, Massachusetts, USA.

Abstract

Issues such as whether a product is easy to learn, to use, and whether the user can efficiently complete tasks using it, may greatly affect a product's acceptance in the marketplace. In software engineering, the support for usability is widely believed to be independent of software architecture design. This belief stems from the efforts to separate the user interface component from the application's internal logic, thus enabling changes to the interface without affecting the software architecture. This assumption has been recently challenged by Bass and colleagues at the Software Engineering Institute who argue that architectural patterns must be in place in order to support good usability design. We investigate the degree to which this revised belief is true with a case study on the redesign of an existing application for better supporting usability. The redesign is based on implementing a task-oriented interface and help system. Preliminary results show that much, though not all, of the required changes can be done without major changes to the software architecture of GIMP. The results do support the idea that the envisioned usability redesign could not easily be implemented without some powerful architectural features that do not necessarily correspond to the specific patterns identified by Bass and his team.